Distributed Commit Protocols for Nested Atomic Actions

by

Sharon Esther Perl

November 1988

This work was supported in part by the National Science Foundation under Grant DCR-8510014, and in part by the Defense Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under Contract N00014-83-K-0125.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TR-431	5. MONITORING ORGANIZATION REPORT NUMBER(S) N00014-83-K-0125
6a. NAME OF PERFORMING ORGANIZATION MIT Laboratory for Computer Science 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research/Department of Navy
6c. ADDRESS (City, State, and ZIP Code) 545 Technology Square Cambridge, MA 02139	7b. ADDRESS (City, State, and ZIP Code) Information Systems Program Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/DOD 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Blvd. Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Distributed Commit Protocols for Nested Atomic Actions

12. PERSONAL AUTHOR(S)
Perl, Sharon Esther

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Technical	FROM _____ TO _____	1988 November	160

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Commit Protocols, Transactions, Nested transactions, Atomicity, Distributed computer systems, Eager diffusion, Argus, I/O automata

19. ABSTRACT (Continue on reverse if necessary and identify by block number)
Nested atomic actions are a useful tool for building robust distributed programs. This thesis examines two aspects of the design of commit protocols for nested actions: semantics and efficiency.

Most existing protocols provide relatively weak guarantees about when sites learn the outcomes of actions. We introduce the notion of an eager diffusion semantics for action completion. A protocol that supports eager diffusion guarantees that a site in a distributed system knows as much about commits and aborts of actions as do the actions running at the site. In particular, if an action requesting a lock on an object knows that the lock is available, perhaps because it observed the commit or abort of the former holder of the lock, then the site managing the lock can grant the request based on purely local information; no communication with other sites is required.

The focus of the work is the design and rigorous correctness proof of a new nested (cont.)

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ☑ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL Judy Little, Publications Coordinator	22b TELEPHONE (Include Area Code) (617) 253-5894 22c. OFFICE SYMBOL

19. action commit protocol that supports eager diffusion. The protocol works by piggybacking information on existing messages flowing around the system. A series of optimizations that reduce the amount of information added to messages and stored at sites lays the groundwork for an efficient implementation of the protocol. The optimizations generalize ad hoc techniques used in existing systems, and the proof defines the assumptions needed to ensure the correctness of the optimizations.

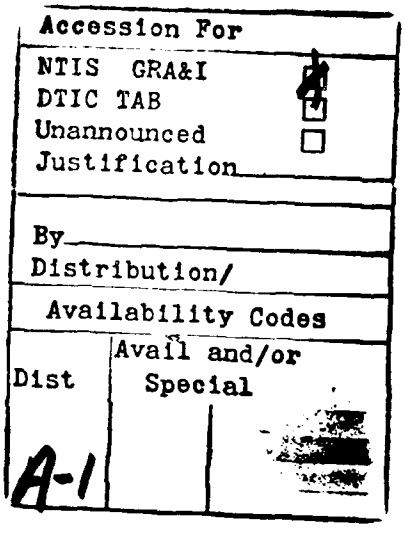
Even the most efficient of the protocols used in existing systems still have performance problems. We suggest how to apply our optimization techniques to improve the performance of nested action commit protocols that provide only a weaker semantics than eager diffusion, of which the protocol used in the Argus distributed system is an example.

Distributed Commit Protocols for Nested Atomic Actions

by

Sharon Esther Perl

November 1988



This work was supported in part by the National Science Foundation under Grant DCR-8510014, and in part by the Defense Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under Contract N00014-83-K-0125.

Distributed Commit Protocols for Nested Atomic Actions

by

Sharon Esther Perl

Abstract. Nested atomic actions are a useful tool for building robust distributed programs. This thesis examines two aspects of the design of commit protocols for nested actions: semantics and efficiency.

Most existing protocols provide relatively weak guarantees about when sites learn the outcomes of actions. We introduce the notion of an *eager diffusion semantics* for action completion. A protocol that supports eager diffusion guarantees that a site in a distributed system knows as much about commits and aborts of actions as do the actions running at the site. In particular, if an action requesting a lock on an object knows that the lock is available, perhaps because it observed the commit or abort of the former holder of the lock, then the site managing the lock can grant the request based on purely local information; no communication with other sites is required.

The focus of the work is the design and rigorous correctness proof of a new nested action commit protocol that supports eager diffusion. The protocol works by piggybacking information on existing messages flowing around the system. A series of optimizations that reduce the amount of information added to messages and stored at sites lays the groundwork for an efficient implementation of the protocol. The optimizations generalize *ad hoc* techniques used in existing systems, and the proof defines the assumptions needed to ensure the correctness of the optimizations.

Even the most efficient of the protocols used in existing systems still have performance problems. We suggest how to apply our optimization techniques to improve the performance of nested action commit protocols that provide only a weaker semantics than eager diffusion, of which the protocol used in the Argus distributed system is an example.

Keywords: Commit Protocols, Transactions, Nested transactions, Atomicity, Distributed computer systems, Eager diffusion, Argus, I/O automata

This report is a modified and extended version of a Master's thesis of the same title, submitted to the Department of Electrical Engineering and Computer Science on 30 October, 1987. The thesis was supervised by Professor William E. Weihl.

Acknowledgements

There are many people who helped make this thesis a reality and who contributed to my enjoyment of the research and writing processes. In particular, my sincere thanks go to:

Bill Weihl, for being a most patient, attentive, encouraging and enthusiastic thesis supervisor, and a good teacher.

Barbara Liskov and past and present members of the Programming Methodology Group, for providing a pleasant research environment and many helpful and stimulating discussions (some even research-related).

Nancy Lynch and members of the Theory of Distributed Systems Group, in particular, Alan Fekete and Mark Tuttle, for helpful discussions about the formal aspects of this work.

Alan Fekete, Gary Leavens, and Brian Oki, for their prompt reading of a draft of the thesis. Their many helpful comments and questions improved the presentation.

My parents, for their constant love and support, and for always believing in me.

Mark Reinhold, my husband, for carefully proofreading a draft of the thesis, for being my personal Unix and TEX wizard, and, in general, for making everything all the more worthwhile.

Contents

List of Figures

Chapter 1

Introduction

Concurrency and failures are two of the significant characteristics that make the programming of distributed systems more difficult than the programming of centralized systems. Nested atomic actions are a useful programming tool for constructing software for reliable distributed systems because they help the programmer to cope with concurrency and failures. Many commit protocols have been designed for single-level action systems. This thesis is concerned with the design of efficient commit protocols for nested actions.

The complexity of reasoning about programs increases significantly when concurrently executing programs may interact and when failures of system components may prevent parts of computations from completing. Single-level atomic actions have long been recognized as a helpful tool for building reliable distributed programs [Lampson 1981]. An *atomic action*—or simply, an *action*—is a unit of computation that is *serializable* and *recoverable*. Serializability means that actions that execute at the same time appear to execute in some non-overlapping order; the correctness of a single action may be determined independently of the other actions in the system. Recoverability means that an action appears to execute either completely (it *commits*) or not at all (it *aborts*); a running action that encounters a failure may abort in order to avoid leaving the system in an inconsistent state.

Atomic actions can be generalized to *nested* atomic actions by using *subactions* to build higher-level actions in a hierarchical fashion, forming trees of nested actions with *topactions* at the roots [Moss 1981]. In talking about nested action systems, we use the term *action* to refer to both topactions and subactions. Subactions are serialized relative to other subactions that share the same parent (their siblings) in the action tree. The commit of a subaction is contingent upon the commit of all of its ancestors up to its topaction; the effects of a subaction are undone if one of its ancestors aborts.

One problem in implementing single-level atomic actions is to ensure that the decision to commit or abort each action is consistent for all sites in the system. This same problem exists in nested action systems for determining the outcomes of topactions. We call a protocol that solves this problem a *toplevel commit protocol.*

The literature abounds with discussions of toplevel commit protocols, a well-known example being *two-phase commit* [Gray 1979].

In nested action systems there is the additional problem of committing and aborting subactions. The requirements for handling subaction completion are less stringent than those for topactions. It is not necessary to ensure that all sites involved in a subaction agree on its outcome ahead of time, since the effects of a subaction may be undone any time before its topaction commits by aborting an ancestor of the subaction. It is necessary, however, for sites involved in the subaction to have the means of learning the outcome when they need the information and for all sites to receive consistent reports about the outcome.

Some designers of nested action systems have taken the view that action completion should be handled uniformly at all levels of the action tree, and thus have used toplevel commit protocols to handle subaction completion as well (*e.g.*, see [Allchin 1983]). While this is valid, it increases the expense of subactions, making them less useful than they might otherwise be. Other designers of nested action systems have taken advantage of the differences between topactions and subactions, introducing more efficient methods for handling subaction completion [Liskov 1984, Spector *et al.* 1987].

1.1 Motivation

Nested actions have been incorporated into distributed programming systems for various research projects (*e.g.*, Argus [Liskov 1984], Camelot [Spector *et al.* 1987], Clouds [Allchin 1983], and Locus [Mueller *et al.* 1983]). Each of these systems employs some protocol for handling nested action completion. Our motivation in undertaking further study of the problem is twofold. First, the protocols used in existing systems vary widely in efficiency; some have proved to be completely impractical while others provide reasonable performance for the prototype systems in which they run. Even in the more successful cases, however, there seem to be opportunities to improve performance by introducing optimizations and taking better advantage of information already present in the systems. Second, most existing protocols that we know of provide only relatively weak guarantees about when sites learn the outcomes of actions. This is true of toplevel commit protocols as well as nested action commit protocols. Experience with distributed applications [Greif *et al.* 1987] has indicated that support for a stronger semantics would aid programmers in their efforts to provide certain types of services.

The Locus and Argus systems provide examples of two extremes in approaches to handling nested action completion with regards to efficiency. The protocol used in an early version of the Locus distributed operating system forces a committing subaction to wait until all participating sites are informed of the commit, thus introducing significant delays at all levels of the action tree. The protocol used in the Argus

programming language and system is based on a kind of lazy evaluation technique, and avoids the delays at action commit and abort time that are present in the Locus protocol. In Argus, when a site needs to know the outcome of an action it queries another site that has the information. However, a single query does not always suffice; many queries may be required to determine the outcome. Communication with remote sites always introduces delay, and under the Argus scheme the delay is introduced exactly at the time when the information is required by the querying site.

Both the Locus and Argus protocols provide relatively weak guarantees about when sites learn of the completions of actions. Specifically, the programmer of an action may not assume that another action is committed or aborted at all objects in the system as soon as it is "observed" to be committed or aborted at some of them. If an action is forced to wait for a lock upon request, this will be noticeable as a delay until the site managing the lock can obtain the necessary information to grant the lock. More importantly, if an action may test the availability of a lock without waiting, then it may be informed that it cannot obtain the lock, contrary to expectation.[1] The Locus protocol supports a slightly stronger semantics than the Argus protocol in that when an action runs sequentially after another action, the second action will observe the first one to be completed at all sites. However, Locus is not able to make this guarantee for actions that run concurrently with one another.

The Locus protocol was eventually judged impractical by its designers. The Argus protocol appears to work acceptably, though there still seem to be optimizations that could reduce the number of query messages and their accompanying delays, as we will discuss in Chapter 4. However, the lack of support for a stronger semantics has presented problems in attempts to build some applications, as evidenced by the problems encountered by the designers of CES [Greif *et al.* 1987].

CES is a distributed collaborative editing system implemented in Argus that allows several authors to cooperate in writing a document. One requirement of CES is that it permit users to edit documents concurrently, while minimizing delays encountered by one user because of another user's activities. Another requirement is that the consistency of the data must be maintained, even in the presence of concurrency and failures. The implementors of CES encountered problems with Argus in trying to provide the desired semantics for a primary data structure, called a *version stack*, that supports concurrent access to document sections.

CES permits an author to establish versions of document sections and to revert to a previous version. At the same time, any author can read the entire document while others are editing parts of it. Only one author can modify a section at a time. If one author has written a tentative version of a section and another tries to read the same section, the reader will get the most recent version. Multiple versions of each

[1] This capability is useful for implementing highly concurrent atomic data types; see Section 2.3.2 for more information.

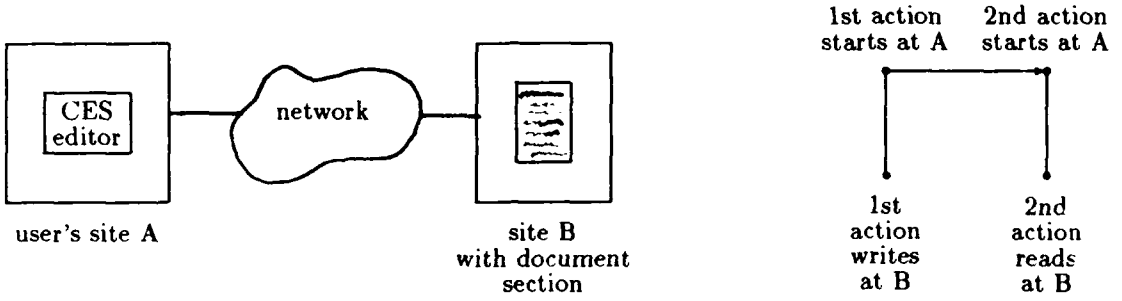


Figure 1.1: A problem with version stacks. The user is at site A editing a document section stored at site B. The relationship between the actions that write and read the section is shown at right. The arrow from the first action to the second means that the first actions runs entirely before the second.

document section are maintained in a stack, with the usual operations provided to push a new version onto the stack, pop a version off the stack, read the top of stack and reset the stack. An additional operation, called *fasttop*, is provided to permit fast response time when authors read sections that are being modified simultaneously by other authors. While the *top* operation always returns the top version on a stack, waiting if necessary until the action currently modifying the version has completed, the *fasttop* operation nondeterministically returns some version that is guaranteed to be no older than one returned in a previous call (unless there has been an intervening *pop* or *reset* operation). The specification of *fasttop* also requires that the version returned be as recent as possible.

The problems encountered in implementing CES were fairly complex. We will consider an example problem similar to one in CES, but simpler, to illustrate the semantic problems with Argus' handling of nested actions. The problem involves providing a reasonable specification for version stacks. The scenario is illustrated in Figure 1.1. Suppose a user is working at site A, and part of the document is stored at site B. The user could make a change to a document section stored at site B in one action. Once that action has committed, the user could use the *fasttop* operation to see that part of the document. If site B does not yet know that the first action committed, the fasttop operation might return an older version of the section. The user knows that the section has been changed and that the modifications have been committed, but until the commit event is known at all sites involved he may see information that is out of date.

This seemingly strange behavior results from the way in which Argus processes commits and aborts. Suppose the two actions run by the CES user are separate topactions. When the first action commits, all locks it holds on objects at site B can

be released. Since the second action runs only after the first action commits, when the second action tries to read those same objects it should be able to obtain any locks that were held by the first action. However, the Argus commit protocol allows the second action to begin executing as soon as the first action's commit is decided at site A, and possibly before other sites learn of the commit. Since site B may not yet have learned that the first action committed, it may not be able to grant the locks to the second action. If the second action only tests whether particular locks are available, without waiting for the site to find out about the commit of the first action, then it will be told that the locks are unavailable. This is what happens when the second action performs the *fasttop* operation. Given the limited guarantees that Argus can make about propagation of commit and abort information, there is no way to implement version stacks to provide a more reasonable behavior for *fasttop*.

A similar problem can also occur in Argus if, instead of peforming the remote accesses to the document as different topactions, the accesses are performed in subactions. This problem is not limited to Argus; it is present in all systems that use commit protocols similar to the one used in Argus. For example, the Camelot system [Spector *et al.* 1987] has the same problem.

In general, the types of applications that could benefit from the eager diffusion semantics (and, accordingly, could suffer from a lazy diffusion semantics) are those in which a high degree of concurrency is achieved through the use of non-determinism in the specification. Eager diffusion allows the application designer to limit the non-determinism in the specification, thereby eliminating some behaviors that might seem suprising or unacceptable to the user. Another example of this problem appears in [Weihl 1984] in the context of a data type specification for a highly concurrent queue, called a *semiqueue*. In Weihl's example, the problem can be solved without our full distributed protocol. However, the semantic issue is the same.

In addition to the problems with semantics, the work on CES also reveals efficiency problems when large numbers of query messages are generated. The Argus implementation has since been modified to eliminate the particular efficiency problems encountered in [Greif *et al.* 1987]. However, the solution is fairly specific to the situations that arose in that work.

1.2 Approach

In the research reported in this thesis we have tried to examine the semantic problems with known commit protocols for nested actions in a systematic and general way. Most existing nested action commit protocols guarantee only that a site will eventually learn the outcome of an action. The strongest guarantee that could possibly be stated for propagation of commit and abort information is that as soon as the outcome of an action is decided at one site, then it is known at every site. Clearly, this is impossible in a distributed system where there are communication delays and sites may be down

at various times. Instead, we introduce the notion of an *eager diffusion semantics* for propagation of commit and abort information and describe a new protocol that supports this semantics. Eager diffusion requires that once an *action* "learns" about the commit or abort of another action at a site, it will retain that knowledge at every other site it visits. Thus, in the CES example above, when the second action is started at site A, it implicitly knows that the first action must have committed, and it will retain that knowledge when it visits site B. This contrasts with the *lazy diffusion semantics* provided by most commit protocols, including those in Argus and Locus. The lazy diffusion semantics is so-named because of the "lazy" way in which information about commits and aborts is diffused through the system. (Argus is the "laziest" in this respect).

For concreteness, we restrict our attention in this thesis to systems that use locking for concurrency control (as Argus does). We make the definition of eager diffusion more specific by requiring that a site have enough local information to grant a lock to a requesting action if the action knows that it ought to be able to obtain the lock. An action's *knowledge* includes its state (data and control information), the programs of potentially all other actions running in the system, as well as all deductions that can be made from this knowledge. A site's *information* includes the states of locks at the site (the current lock holders and requestors), as well as explicit information that has been recorded at the site about commits and aborts of actions. A site does not attempt to interpret an action's state or program. The job of a protocol that supports eager diffusion is to ensure that a site's information about availability of local locks is at least as complete as the knowledge of each action running at the site. Chapter 3 describes in detail how an action can know that it ought to be able to obtain a lock. Although we have not yet studied the problem carefully, we believe that it is possible to develop reasonable interpretations for the eager diffusion semantics in systems that use other methods of concurrency control, such as optimistic schemes or timestamp schemes (*e.g.*, [Kung & Robinson 1981, Reed 1978, Weihl 1984]).

The protocol that we have designed to support eager diffusion works by piggybacking information about commits and aborts of actions on messages that already flow around the system (*e.g.*, remote procedure call and reply messages and toplevel commit messages). Actions spread through the system via such messages and the protocol essentially guarantees that everything that an action knows is carried with it in its travels. As the basis for a practical implementation of such a protocol we introduce several optimizations that allow us to reduce the amount of explicit information that must be added to messages. One optimization takes advantage of information about action trees encoded in *action identifiers* (the names of actions) to make inferences about which other actions must have committed or aborted. Another optimization uses the absence of information about aborts to infer that particular commits must have occurred.

Our proposed protocol does not subsume other protocols that support the lazy

diffusion semantics, but rather works in tandem with them. Our protocol guarantees that a site knows as much as an action requesting a lock at the site knows. Eager diffusion does not address the case where an action does not know that it can get a lock. In that case, we resort to some other protocol to ensure that the site eventually obtains enough information to release the lock.

We have not yet implemented our protocol. Thus we do not have any concrete results about its practicality. There is a problem of garbage collecting the commit and abort information that must be maintained at each site. Provided that this problem can be solved, we do believe that our optimizations can limit the amount of information added to messages enough that the additional information should not pose efficiency problems in most cases. We suggest one solution to the garbage collection problem in Chapter 4, but we are not yet convinced that the solution is practical. There may be other, more practical solutions.

Whether or not our protocol to support eager diffusion proves to be practical, we have found that a number of the optimizations developed for our protocol could also be incorporated into Argus (and similar systems) to improve its performance in supporting lazy diffusion. We describe this in detail in Chapter 4.

1.3 Related Work

Much of the initial inspiration for our nested action commit protocol derives from work on an orphan detection algorithm for Argus [Liskov *et al.* 1987c, Walker 1984]. This algorithm piggybacks information on existing messages in the same way that our protocol does, although the actual information is somewhat different. The idea of using the absence of abort information to infer commits is used in the Argus implementation to reconstruct portions of action trees based on abort information passed up the tree as actions commit [Liskov *et al.* 1987a]. It is similar in flavor to the *presumed abort* and *presumed commit* variations on two-phase commit described in [Mohan & Lindsay 1983]. The scheme that we suggest for garbage collecting commit and abort information in our protocol is based on one developed for Argus orphan detection. [Liskov *et al.* 1987c].

Not surprisingly, much of the formal work on proving the correctness of our protocol borrows from work on proving the correctness of the Argus abort orphan algorithm, presented in [Herlihy *et al.* 1987]. The part of the proof in Appendix B also uses ideas from [Fekete *et al.* 1988].

The problem of designing commit protocols for toplevel actions, where the goal is to reach agreement on outcomes of actions among all involved sites, has been a fertile area of research for some time. For example, see [Gray 1979], [Mohan & Lindsay 1983], [Lampson 1981], [Lindsay *et al.* 1979], and [Lindsay *et al.* 1984] for descriptions of the two-phase commit protocol and its variations, and [Skeen 1981] and [Dwork & Skeen 1983] for descriptions of non-blocking

and three-phase commit protocols. Note that the "nested two-phase commit protocol" described in [Gray 1979] is also a toplevel commit protocol; the term "nesting" in that context describes the pattern of communication among participants and has nothing to do with nested actions.

There have been a number of research projects involving the design and implementation of distributed programming systems incorporating nested actions. Some of the more prominent ones are:

- Argus [Liskov *et al.* 1987a]. Argus's handling of nested action commit will be described in detail in Chapter 2.

- Locus [Mueller *et al.* 1983, Weinstein *et al.* 1985]. As we have already mentioned, early versions of Locus introduced significant delays at each level of the action tree when committing or aborting nested actions. Later versions of Locus do not have true nested actions, apparently because the earlier implementation of nested actions in Locus proved too expensive.

- Camelot [Spector *et al.* 1987, Spector & Swedlow 1987]. Essentially, the Argus nested action commit protocol is used. Camelot's model of nested actions is slightly different from Argus's—for example, Camelot allows individual actions to be distributed—and the protocol is modified, as necessary, to accommodate the differences. The designers of Camelot have recently begun looking into possibilities of piggybacking information on messages in order to reduce lock propagation queries [Duchamp 1987].

- Clouds [Allchin 1983, Kenley 1986]. A two-phase commit protocol is used to commit each nested action. The Clouds designers have taken the view that the commit protocol should be as uniform as possible at all levels of the action tree. Unlike Argus (and like Camelot), Clouds does not require remote calls to be performed in new subactions. Thus the programmer is not forced to introduce subactions as frequently as in Argus. However, making subactions more expensive limits their usefulness.

- ISIS [Joseph & Birman 1986, Birman & Joseph 1987]. The second (most recent) version of ISIS abandoned nested actions. The first version of ISIS (which we refer to as ISIS-1) is based on nested actions, though in a somewhat restricted form since it does not allow sibling actions to run concurrently. Subaction commit and abort in ISIS-1 is handled by broadcasting the outcome to all participant sites at each level of the action tree using the ISIS-1 *obcast* broadcast primitive. Because of the ordering properties guaranteed by *obcast*, returns of remote procedure calls need not be delayed while the outcome of the called action is being broadcast. Although it is difficult to tell from the papers, it seems

that ISIS-1 does support the eager diffusion semantics, at least for information about commits. It is not clear what happens in the case of aborts, where the set of participating sites may not be known by the process sending the abort notification. The work involved in piggybacking commit and abort information (actually, broadcast messages in general) and ensuring that information arrives in an order consistent with the causal relationships of messages is handled by the underlying broadcast primitives. It is worth noting that ISIS-1 also has a problem of distributed garbage collection of the information sent around on *obcast* messages. Their problem is somewhat more amenable to efficient solution than ours because they can detect when information may be discarded. Issues of garbage collection in our protocol are discussed in Chapter 4.

1.4 Roadmap

The remainder of the thesis is organized as follows. Before presenting our protocol, we describe informally, in Chapter 2, the model of nested action systems upon which we base our protocol. Chapter 3 presents a simplified version of the protocol that supports eager diffusion, along with informal correctness arguments. Chapter 4 describes optimizations that can be applied to the simplified protocol to make it practical. The chapter also discusses other efficiency issues, including the garbage collection problem and interactions with orphan detection, and it describes how to adapt some of the optimizations to improve protocols that support only lazy diffusion. In Chapter 6 we give a rigorous correctness proof (in the Lynch-Merritt formal model of nested transaction systems) for the simplified protocol presented in Chapter 3. Chapter 7 summarizes and concludes the thesis, with mention of further work that could be done. The two appendices supplement the proof in Chapter 6.

Chapter 2

The System Model

This chapter describes the model of distributed computation upon which we build our commit protocols. The model we employ is that of the Argus programming language and system [Liskov *et al.* 1987a, Liskov 1984, Liskov & Scheifler 1983], although many of the properties that we rely upon are not unique to Argus. We have chosen to provide concrete descriptions of properties of the Argus nested action model that may be less general than actually required, in the hope that this will help the reader to envision more clearly the type of system we have in mind.

2.1 Low-level System

At a low level of abstraction, a distributed system is composed of a collection of nodes connected by a communications network. Distinct nodes communicate with each other only by sending messages (of unrestricted length) over the network. Nodes are typically individual computers, and may be uniprocessors, multiprocessors, time-sharing machines, single-user workstations, *etc.* All components of the system may fail. Nodes may crash or otherwise fail, although we do assume that when a node fails any messages it sends are detectably invalid. The network may fail by partitioning, and messages may be lost, duplicated, delayed, or delivered out of order. We assume that failures are eventually repaired; nodes eventually recover from crashes and partitions are eventually mended. Nodes have access to both stable and volatile storage. Volatile storage is lost in a crash, while stable storage is intact upon the recovery of a node [Lampson 1981].

2.2 Guardians

At a high level of abstraction, we view a distributed system as a collection of *guardians*. A guardian is an active entity that encapsulates and provides access to one or more resources. Each guardian provides *handlers*, which are similar to abstract data type operations, that other guardians may call to access the guardian's resources. Contained inside each guardian are dynamic collections of processes and abstract data

objects. Processes within a guardian may access the guardian's resources and manipulate the guardian's objects directly, without going through handlers.

A guardian resides at a single node in the network, although many guardians may reside at the same node. Each guardian has its own separate address space. Objects that are provided as arguments or results of handler calls are passed by value; a guardian may not have a direct pointer into another guardian's address space. Guardians themselves, and their handlers, are the only entities in the system that are passed by reference in remote calls and that may actually be shared among processes in different guardians.

Guardians are *resilient*; they survive crashes of their nodes with high probability. The objects in a guardian are designated as either *stable* or *volatile*. Stable objects survive crashes with very high probability and are recovered when the guardian is restarted after a crash. Volatile objects are lost in a crash. Thus the permanent state of a guardian must be kept in stable objects. Volatile objects are used to store redundant information (*e.g.*, an index into a database) or information that can be discarded in a crash (*e.g.*, internal process state). All objects in a guardian are stored in a single garbage-collected heap in volatile memory. Stable objects are written to stable storage devices as necessary.

Processes in a guardian are created dynamically, as needed, to service incomming handler calls and to perform background tasks. All processes have access to the guardian's heap, but each process has its own execution stack. The guardian is responsible for scheduling its processes; the processes will timeshare a single processor, while on a multiprocessor they may run concurrently.

Handlers are invoked using *remote procedure call* (RPC). A caller initiates an RPC by sending a call message to the guardian of the handler being called and then waits for a reply message before proceeding. The call message contains the arguments for the handler. The guardian receiving the message runs the call and then sends back a reply message with the results. We assume that RPCs have a *zero or once semantics*. That is, a call either runs exactly once or appears as if it never ran at all. A handler will never execute only partially at the called guardian. This semantics may be ensured, even in the presence of failures, by using actions.

2.3 Actions and Objects

Actions (also called *transactions*) are a mechanism for maintaining consistency of data in the presence of concurrency and failures. While actions are important in centralized systems, and have been used in database work for quite some time, they are particularly useful in distributed systems where the failure modes are more numerous and complex. Programs that modify objects at multiple guardians may partially fail (if one guardian crashes) while other parts of the program continue to run. Communication failures may prevent programs from running to completion after they have

already modified objects. Actions provide a means for the programmer to cope with these failures.

Actions delimit computations and run in processes at guardians. The two properties of actions that ensure that objects manipulated by the computations can be kept consistent in the presence of concurrency and failures are known as *serializability* and *recoverability* (or *totality*). Serializability means that when actions are executed concurrently, the effect is as if they were run sequentially in some order. This allows the programmer of an action to ignore concurrency, simplifying the reasoning about manipulations of objects. Recoverability means that an action either runs successfully to completion (it *commits*) or else has no effect at all (it *aborts*). In the event that a failure occurs during an action, the action may simply abort, undoing all modifications that the computation has made thus far. When an action commits, the system ensures (with very high probability) that its effects will not be lost due to failures.

The abstract *objects* inside guardians encapsulate data. The system provides a number of built-in abstract data types, as well as facilities for building user-defined abstract types. Actions and objects work together, as described below, to ensure atomicity.

2.3.1 Nested Actions

Our model permits actions to be *nested*. A nested action is one that is started from inside another action. Actions may be nested to arbitrary levels, forming action trees with *topactions* at the roots. Actions that are not topactions are called *nested actions* or *subactions*. We use standard tree terminology in referring to the relationships between actions, for example, *parent*, *child*, *ancestor*, and *descendant*. We define an action to be its own ancestor and descendant, for convenience. An action's *proper ancestors* (or *proper descendants*) are all ancestors (or descendants) of the action, not including the action itself. We think of an action as containing all of its descendants. Thus we will often talk about an "action" knowing a fact or running at a site when we mean "a descendant of the action." The use should be clear from context.

Nested actions are particularly useful as a means to obtain checkpoints and concurrency within actions. A subaction may be aborted without causing its parent action to abort. Thus, the parent may create a subaction to attempt some computation, and if that computation fails, either go on with other computations or create another subaction to retry the computation. In this way, the parent's state at the time it starts the subaction is checkpointed[1]; the parent may perform further computation, through the subaction, and still return to its previous state (by aborting the subaction) if that further computation fails.

Concurrency within an action is obtained by allowing a parent to start concurrent

[1]The term *checkpoint*, as used here, does not imply that state is saved in a way that survives crashes.

subactions. While a child action is running, its parent is suspended. However, sibling subactions may execute concurrently. Siblings are serialized at each level of the action tree. Thus there are no problems with concurrent siblings interfering with one another. We refer to an action with concurrent siblings as a *concurrent* action. Actions that do not have concurrent siblings are called *sequential*. Two siblings that run concurrently with each other are called *concurrent siblings*, and ones that run sequentially with respect to each other are called *sequential siblings* (a sequential sibling of an action may be a concurrent action itself). Sequential siblings are ordered according to when they run. A *prior sequential sibling* (or *later sequential sibling*) of an action is a sibling that runs entirely before (or after) the action.

The commit of a subaction is always relative to its parent. If a subaction commits and its parent aborts, the effects of the subaction will be undone. When a subaction T and all its ancestors up to its topaction commit, we say that T has *committed to the top*. When T's topaction then commits we say that T has *committed through the top*.

Another type of nested action, called a *nested topaction*, is sometimes useful. A nested topaction is a topaction that is started from within some other action. A nested topaction differs from a subaction in that the commit of a nested topaction is independent of the commit of its parent and it is serialized relative to the topaction of its parent. The commit of a nested topaction must be handled with a toplevel commit protocol, since the results of the action become permanent upon its commit. Nested topactions are used for performing *benevolent side-effects*, that is, computations whose effects should persist even if the effects of the parent action are eventually undone.

Actions are named by *action identifiers*. An action identifier is structured and contains information about the action tree of the action it describes. In particular, an action identifier for an action contains a name for the action, the guardian at which the action runs, and, in addition, the names of each of the action's ancestors and the guardians where they run. Thus action identifiers are unbounded in size. From the action identifiers for two actions it is possible to determine whether the actions are related (*i.e.*, descendants of the same topaction) and, if so, whether they are descendants of sequential siblings or concurrent siblings.[2] In our protocol we will make use of the action tree information captured in action identifiers.

We mentioned in the previous section that the RPC semantics of *zero or once* execution is accomplished using actions. Each action runs at exactly one guardian; therefore, individual actions are not distributed. However, an action may create nested actions that run at guardians other than its own. An RPC is performed from inside an action as follows. First a *call action* is created as a subaction of the calling

[2]Action identifiers in the current Argus implementation do not differentiate two actions that are themselves concurrent but run sequentially with one another from two actions that run concurrently with one another. It would not be hard to change this.

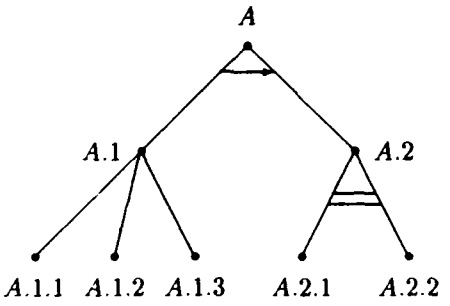


Figure 2.1: An action tree.

action at the guardian of the caller. This call action creates its own subaction, the *handler action*, at the called site. The handler action runs the code of the remote call, completing by committing or aborting. If the handler action aborts, it is as if the call never ran at all. If the handler action commits up to the call action, and it in turn commits to the caller, then the RPC has happened exactly once. The introduction of two subactions to perform an RPC makes it possible for either side of the remote call to abort unilaterally if it encounters failures. For example, if the call action does not receive a reply within a reasonable amount of time it may decide to abort, causing modifications made by the handler action (which is the call action's child) to be undone eventually. Similarly, the handler action may decide to abort if the results cannot be communicated back to the caller or if the guardian executing the handler action encounters problems.

We depict relationships among actions pictorially in action trees, as in Figure 2.1. Nodes represent actions; they are labelled with the names of actions and, sometimes, with the guardian at which the action runs. Branches show connections between actions; children are drawn below their parents. In some examples, we wish to emphasize that particular siblings run sequentially or concurrently with one another. In the figure, the arrow from $A.1$'s branch to $A.2$'s branch indicates that $A.1$ runs sequentially before $A.2$. The parallel lines connecting the branches to $A.2.1$ and $A.2.2$ indicate that these two actions run concurrently with one another. When the ordering of the actions is not important, we omit the indication from the picture.

2.3.2 Atomic Objects

Objects inside guardians are of two kinds: atomic and non-atomic. Atomicity of actions is achieved by using *atomic objects*. If all objects shared by a collection of actions are atomic, then the actions are guaranteed to be serializable and recoverable. Each atomic object does its part to ensure atomicity of actions by performing local concurrency control and recovery.

We assume that atomic objects synchronize the actions that access them using locks. (In our discussions we mainly consider read/write locking, but the statements should extend to any general conflict-based locking scheme.) Thus, before an action can perform an operation on an object, it obtains a lock on the object in the appropriate mode (according to whether it only reads the data or modifies it as well). Any number of actions may simultaneously hold read locks on an object but only a single action, and its ancestors, may hold write locks on an object at a given time. That is, an action T may obtain a write lock on an object if all current holders of read and write locks on the object are ancestors of T, and T may obtain a read lock if all current holders of write locks are ancestors of T.

For our purposes in describing commit protocols, the choice of recovery method is not really important. However, for concreteness we may assume that recovery is performed using versions. When an action first modifies an object, a new copy of the object, called a *version*, is created and the modifications are made to the version. All later reads by the action are performed on the action's version.

When a subaction that holds a lock commits, its lock and version, if there is one, are *propagated* up to its parent; the parent automatically becomes the new logical holder of the lock (see below). When an action aborts, all locks held by it are released and all versions written by it are discarded. When a topaction commits, each version written by it is installed as the new value of the object. (It is at the time of topaction commit that stable objects must be copied from volatile memory to a stable storage device).

We said that a parent automatically becomes the new *logical* holder of a lock when one of its subactions commits. In fact, it may take some time for the information about the commit of the subaction to reach the guardian of the object for which the lock is held. When the information does reach the object's guardian, the parent can actually be recorded as the current holder of the lock. A useful notion in dealing with requests for locks is that of *visibility*. Suppose that an action T is requesting a lock that is currently held in a conflicting mode by an action T'. We say that T' is *visible* to T if T' is committed up to its least common ancestor with T. This implies that the least common ancestor is the current logical holder of the lock, and thus the lock may be propagated to it and then granted to T.

All bookkeeping information associated with an atomic object is maintained by the guardian at which the object is located. This includes versions of the object, as well as records of current lockholders.

The system provides a number of built-in atomic types. It also provides a mechanism for users to define new abstract atomic types that may possibly allow more concurrency than the built-in types. In order to do this, it is necessary to allow the programmer of a user-defined atomic type to "step outside" the action system. Non-atomic objects are provided for this purpose. When an action modifies a non-atomic object, the effects of the modification persist even if the action aborts. Also,

non-atomic objects do not perform synchronization and thus an action could "see" another concurrently executing action that accesses the same non-atomic data. The system also provides low-level mutual exclusion primitives for the programmer of a user-defined atomic type, as well as a means to test whether a lock on an atomic object would be available to an action, without actually obtaining the lock. By combining atomic and non-atomic data, mutual exclusion, lock testing, and careful programming, a programmer can define a new atomic type that will preserve atomicity of the actions using it while providing a greater degree of concurrency than that provided by the built-in types [Weihl 1984].

2.3.3 Action Completion

In order to commit a topaction, the system must execute a toplevel commit protocol to ensure that the action either commits everywhere or aborts everywhere. We assume that the two-phase commit protocol of [Liskov *et al.* 1987a] is used, and have tailored our later discussions to this assumption. However, there is no reason in principle why our commit protocols for nested actions could not be made to work with other toplevel commit protocols.

We briefly review the two-phase commit protocol below. More thorough discussions of two-phase commit in general and two-phase commit in nested action systems may be found in [Gray 1979] and [Liskov *et al.* 1987a].

The participants in the two-phase commit for a topaction are the guardians of those descendants of the topaction that committed to the top. The coordinator is the guardian where the topaction was created. In the first phase the coordinator sends *prepare* messages to all participants. Each participant records the versions written by the preparing action to stable storage, writes a prepare record to stable storage, and then responds "ok"; if the participant is unable to record the necessary information, then it responds "refused."

If all participants respond "ok," the coordinator records the commit of the topaction in stable storage and enters phase two. In the second phase it notifies all participants to commit the action. When a participant receives a commit message, it records the commit on stable storage and then installs the action's versions and releases its locks. If a participant refuses during the first phase, or does not respond, the coordinator aborts the action and then attempts to notify the participants of the abort. If a participant receives such notification, it records the abort on stable storage and discards the action's versions and releases its locks. Notification of an abort is not guaranteed to reach all participants. A participant may determine the final outcome of a two-phase commit in which the topaction aborts using the same query mechanism that is provided to handle subaction commits, described below.

There are two points about two-phase commit that must be noted. A common optimization in implementations of two-phase commit is to allow participants to re-

lease read locks early, when they receive prepare messages in phase one, rather than requiring read locks to be held until notification of the final decision arrives during phase two. A write lock must be held until the outcome is known because the outcome will determine whether new versions of objects are installed or discarded. Read locks will simply be released no matter what the decision is, so it is most efficient to release them as soon as it is known that the action holding the locks has completed its computation. This optimization interacts with the nested action commit protocol that we will present in the following chapters. To simplify the presentation we initially assume that this optimization is not performed. In Chapter 4 we will describe how to relax this assumption and modify our protocol to allow the early release of read locks.

The second point to note concerns topactions that are run sequentially in the same process. Another kind of optimization of two-phase commit is to allow a later sequential topaction to be started in a process as soon as an earlier sequential topaction has finished the first phase of its two-phase commit and the decision to commit or abort it has been made. We do allow this optimization to be used.

We now describe a way to handle commits and aborts of nested actions. The mechanism that we describe here ensures only the lazy diffusion semantics. We will build our protocols on top of this mechanism and fall back on it when convenient. Again, we give only a brief description. More information about this method of committing nested actions may be found in [Liskov *et al.* 1987a].

Commits and aborts of nested actions do not require a two-phase commit protocol. If a decision is made to commit a subaction and it turns out that not all guardians visited by the subaction are able to carry out the commit, then the subaction may be effectively aborted by aborting one of its ancestors. Thus the guardian where a subaction runs decides independently to commit or abort the subaction and notifies only the subaction's parent of its decision via the RPC reply message. Compared to the commit protocol for topactions, very little delay is introduced in the process of committing or aborting a subaction. However, this means that other guardians visited by the subaction may still be holding locks on behalf of the subaction even after it commits or aborts.

If a lock held on behalf of one action is needed by another action, the guardian where the lock is held can query to determine whether the lock can be released or propagated. The query is typically directed to the guardian of the action that is the least common ancestor of the action holding the lock and the action requesting the lock. If that ancestor has aborted, or if it is known that the lock holder did not commit to the least common ancestor, then the reply message will indicate that the lock can be released. Otherwise, the guardian where the lock is held could query to other guardians that may have information about the status of the lock requestor, or it could simply wait and try querying to the least common ancestor at a later time. When a guardian receives such a *lock propagation query* it may send a *lock*

propagation query response message containing the reply or an indication that it does not know the answer. A guardian that ran a subaction may decide to send *courtesy* commit or abort messages to other guardians where the subaction is known to have visited. These are like unsolicited lock propagation query responses. We refer to both courtesy messages and query response messages, as well as two-phase commit messages containing commit or abort information, as *lock propagation messages.*

2.3.4 Orphans

An *orphan* is a computation that continues to run even though its results are no longer needed. Orphans arise in two ways: from aborts and from crashes. For example, the caller of a remote subaction may abort or its guardian may crash, leaving the remote subaction running as an orphaned action. Also, a guardian holding locks on behalf of some remote action may crash while the action is still active, implying that the action will never be able to commit through the top. In both cases, there is no point in allowing the orphan to continue running since it or one of its ancestors will eventually abort.

Orphans are undesirable for two main reasons: they waste resources (by wasting processor time and holding locks that prevent other computations from proceeding) and they may see inconsistent information. In an action system we need not worry that orphans will corrupt permanent data because they will never be permitted to commit through the top and thus their effects will eventually be undone. However, when an orphan is abandoned by its caller, locks that the orphan assumes are still held on its behalf may be released. This means that assumptions implicit in the orphan's program may be violated, possibly causing unexpected program behavior (for example, infinite loops or incorrect output).

Orphan detection (or *orphan destruction*) is the process of determining that computations are orphans and killing them off. A desirable property of an orphan detection method is that it identify and kill orphaned actions before the orphans see inconsistent states and before they waste important resources. Typically, different methods must be used to detect abort orphans and crash orphans. A number of orphan detection schemes have been proposed (*e.g.*, see [Liskov *et al.* 1987c, McKendry & Herlihy 1987, Duchamp & Spector 1987]) and we will not describe them here. We discuss the interactions of orphan detection with our protocol in Chapter 4.

2.4 Summary

In summary, the following characteristics of nested action systems are assumed in the descriptions of our protocol:

- Active entities, called *guardians*, manage objects and nested actions. Each action and object is local to some guardian.

- Guardians support the creation of remote subactions through *remote procedure calls.* Remote subactions are created by sending an RPC call message to the guardian at which the action is to run. The results of the action, including its completion status, are returned to the caller in an RPC reply message. The action management facility in a guardian has access to all incoming and outgoing RPC messages, and may piggyback unbounded amounts of information on these messages.

- An action identifier contains the identities and home guardians of all ancestors of the action it names. It is possible to compare two action identifiers to determine whether the actions they describe are descendants of concurrent siblings or sequential siblings; in the latter case, the action identifiers also indicate the ordering of the actions.

- Concurrency control is accomplished using strict two-phase read/write locking [Gray *et al.* 1976].

- A query mechanism exists to allow sites to determine the outcomes of actions. A site wishing to determine the outcome of an action sends lock propagation query messages and receives lock propagation query response messages. The query mechanism guarantees only that the information returned in lock propagation query responses is correct. No time bound on responses is assumed.

- The variation of the two-phase commit protocol employed in the Argus system is used to commit topactions, with one exception. For most of our discussion we do not allow read-locks to be released until after phase two has completed; this restriction is relaxed in Chapter 4. Once a topaction has completed the first phase of two-phase commit and its outcome has been determined, the next sequential topaction in the same process may start executing.

- All messages that convey the completion status of an action are referred to as *lock propagation messages.* These include lock propagation reply messages, courtesy commit and abort messages, and two phase commit messages that contain this information.

- Crash orphan detection is performed.

- Objects in guardians are of two kinds: atomic and non-atomic. Our protocol will only guarantee the eager diffusion semantics for information that actions obtain through the action tree and through atomic objects.

Chapter 3

A Commit Protocol for Nested Actions

We now present a new commit protocol for nested actions that supports the eager diffusion semantics for action completion. The protocol ensures that if an action "knows" that it ought to be able to access an object, then the system will have enough information to permit the access to occur. It works by piggybacking commit and abort information on messages that are already flowing around the system. The full protocol is highly optimized to reduce the quantity of data that must be added to messages. In this chapter we present a simplified version of the protocol. The next chapter describes optimizations and discusses efficiency issues.

The protocol described in the following sections is actually only a partial protocol—it is the part that ensures that commit and abort information is spread through the system in time to guarantee the eager diffusion semantics. There will be cases where an action accessing an object has no reason to know whether or not the access should succeed immediately, and the system does not happen to have the necessary information available. In these cases we rely on some other mechanism, such as Argus's lock propagation queries, to supply the information. We also assume that completion of topactions is handled by some standard toplevel commit protocol. Our descriptions are tailored to two-phase commit but could be adapted to work with other top-level commit protocols.

Before we can consider designing a protocol for the eager diffusion semantics, we need a more precise definition of an action's knowledge. How can an action "know" about the execution status of another action? In Section 3.1 we address this question. Section 3.2 provides a series of examples that illustrate what actions can know. Then in Sections 3.3 and 3.4 we present our simplified protocol, and in Section 3.5, argue informally for its correctness. This version of the protocol has obvious efficiency problems but is relatively easy to understand. In Section 3.6 we return to the examples presented in the earlier section, explaining how the protocol works to guarantee the eager diffusion semantics in each case.

3.1 Actions' Knowledge

We want to design a protocol that ensures the eager diffusion semantics, *i.e.*, if an action "knows" that it should be able to get a lock, then the guardian of the object being locked will have enough local information to give the lock to the action. To do this we must define more precisely what an action can know that allows it to infer that a lock should be available and how it can know it. We keep the discussion informal in this chapter in an attempt to provide the reader with enough intuition to understand the protocol without getting lost in details. Chapter 6 presents a correctness proof in a formal setting.

Under the usual definition of knowledge in distributed systems, a process *knows* a predicate P in an execution if P is true in every execution of the system that looks the same to the process [Halpern & Moses 1987]. By "looks the same," we mean that the process has the same local state. In terms of our problem, an action knows that a lock should be available in a particular execution if the lock should be available to the action in every execution in which the local state of the action is the same.

A lock will be available to an action only if all holders of conflicting locks are ancestors of the action. Thus, if an action knows that a lock should be available, then, in every execution that looks the same to the action, at least one of the following conditions must be true of every potentially conflicting lock holder:

C1: It must be committed up to one of the lock requestor's ancestors, or

C2: It must have an aborted ancestor (it is *effectively aborted*), or

C3: It must never have obtained the lock.

Furthermore, the requesting action must be able to determine from its state that at least one of these conditions is true—otherwise there would be executions that look the same to the action in which none of the conditions are true.

The next question, then, is how can an action acquire information in its local state about a potentially conflicting lock holder that would imply one of the conditions above? There are three kinds of information about the execution status of another action that an action can have. It can have information that an action committed, that an action aborted, or that an action effectively never accessed an object. (By "effectively never accessed" we mean that either it never accessed the object, or it did access the object but the abort of an ancestor removed the effects of the access.) Thus, an action can acquire information in its local state specifically allowing it to know that condition C1 holds or that condition C2 holds. Also an action can acquire information about an effective non-access, allowing it to know that at least one of C2 or C3 holds. The sources of information may be direct or indirect, as described below.

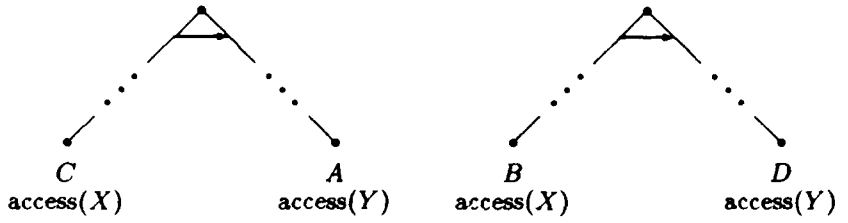


Figure 3.1: Implications of effective non-access.

There are two direct sources of information about the execution status of another action available to an action A:

1. Children: A can learn about its children's commits and aborts.

2. Objects: By accessing an object X, A can learn about commits of all visible actions that held locks on X. Also, A can learn that particular actions have effectively not accessed X before A accessed X. This in turn can imply that other actions have effectively not accessed other objects.

The *visibility* condition for commits at objects is related to the locking rules. Recall that an action B is visible to A if B has committed up to its least common ancestor with A. This is exactly the condition required for a lock held by B (when B does not abort) to be given to A; the lock must first be propagated to an ancestor of A. Note also that an action's committed child is visible to the action. Thus all commits that an action directly knows about are for actions visible to it.

To understand how an action can know that another action effectively never accessed an object, consider the following example. Actions A and B run concurrently with each other, and both read and write an object X. Each action records at X the fact of its own commit. If the first action to access X commits up to its least common ancestor with the other action, the data that it wrote becomes accessible to the other action. If an ancestor of the first action that wrote data aborts, the effects of the access will never be observable to the other action. Thus, it is possible for each action to determine, when it obtains the lock on X, whether the other action has effectively never accessed X. Notice that we do not say that an action can directly learn of the abort of another action that accessed an object. Based on information obtainable through the object, when an action observes that another action effectively never accessed the object, there is no way for it to distinguish the case of the action running and having its effects undone from the case where the action had not yet run.

The implications of effective non-accesses, mentioned in item 2 above, arise because of the way that actions may be structured. It is possible to write a program

in such a way that if action B has effectively not accessed object X by the time action C accesses X, then action D will have effectively never accessed object Y before action A accesses Y (see Figure 3.1). If action A knows, through action C, that B has effectively never accessed X, then A could expect to get a lock on Y since D effectively never accessed Y. For the effective non-access of B to imply the effective non-access of D, the following must be true:

1. If B does not access X before C, then D does not access Y before A.

2. If B does access X before C and is effectively aborted, then either D will not access Y by the time A does, or D is also effectively aborted by the same action that caused B to be effectively aborted.

If these conditions were not true, then we hypothesize that A could not know the implication in the first place.[1]

An action can also obtain information indirectly:

1. Through its parent. An action can know whatever its parent knew at the time the parent created the action.

2. Through committed actions that are known to it. If an action knows that some other action is committed, it can know whatever the committed action knew at the time it committed.

An action cannot learn what an aborted action knew, other than what the aborted action could have learned from its parent at the time it was created. If an action A knows about the abort of another action B, then A does know that B's parent attempted to create B. Since the parent's attempt to create B could be conditional upon everything that the parent knows, it is possible that A can know whatever B's parent knew when it created B. However, A will not necessarily know anything else that was known to B for the following reasons.

- Because actions may be aborted at any time, no action can infer that particular events must have taken place simply based on the fact that some action aborted. While an action may specifically decide to commit or not based upon what it knows, an action does not have total control over its own abort.

- When an action that modifies an object is effectively aborted its modifications are undone and so are not available to any later action accessing the object.

[1] We do not know how to argue informally that this claim is true, although it appears to be so after careful thought. This issue does not arise in our formal definitions in Chapter 5, where we take a somewhat different approach.

- When a child of an action aborts, the child may not return any information to its parent that it learned while running.[2]

Based on the above discussion, we can characterize an action A's knowledge of the execution status of another action.

- Commits: Visibility is transitive; any action visible to an action visible to A is also visible to A. Also, A's ancestors are defined to be visible to A. Thus, it is not hard to see that all commits known to A are for actions visible to A.
- Aborts: A knows about aborts that actions visible to A knew about, and it knows about aborts of its children. Thus A knows about the aborts of children of actions that are visible to A, as well as aborts of its own children.
- Effective non-access: We cannot easily classify those actions that A may know to have effectively never accessed an object. When A is granted a lock that was held by an effectively aborted action, it knows that the action effectively never accessed the object. This effectively aborted action need not bear any particular relationship to A or to the actions visible to A. The same is true for actions whose effective non-accesses A knows about by implication.

3.2 Examples of Knowledge

Some examples will help to illustrate how an action can acquire information about the status of another action that allows it to "know" that it should be able to get a lock. We show three examples, one for each kind of information that an action can have. In each example, we describe the actions and objects involved; we assume that there are no other actions in the system that are interested in the particular objects.

3.2.1 Knowledge of a Commit

Our first example illustrates the case of an action knowing about the commit of another action. In this case, the action learns about the commit from its parent.

The action tree in Figure 3.2 shows an action A that has two sequential remote children, $A.1$ and $A.2$.[3] A runs at guardian G_1 and its children run at guardian G_2. First, $A.1$ runs, writes object X at G_2 and commits up to A. Then $A.2$ runs and requests a read lock on X. A knows about $A.1$'s commit, and since $A.2$ runs sequentially after $A.1$ it can know about $A.1$'s commit through information received from A.

[2] In Argus, programmers are free to violate this restriction and face the consequences. Unless care is taken, serializability may be lost [Liskov *et al.* 1987b, Appendix III].

[3] Actually, in Argus each of these children would be handler actions and there would be call actions between them and their parent, but this is not important in the example.

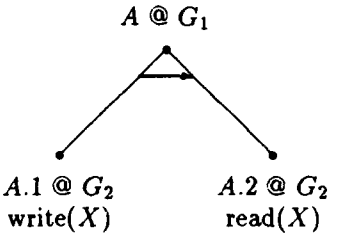


Figure 3.2: $A.2$ knows that $A.1$ (a prior sequential sibling) committed up to A.

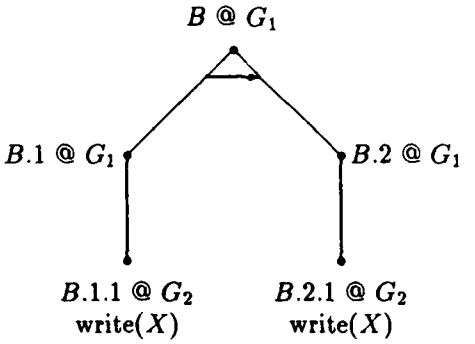


Figure 3.3: $B.2.1$ knows that $B.1.1$ is effectively aborted since it knows that $B.1$ aborted.

3.2.2 Knowledge of an Abort

The second example illustrates the case of an action knowing about another action's abort. Figure 3.3 shows an action B that has two local sequential children, $B.1$ and $B.2$, each of which has a remote child $B.1.1$ and $B.2.1$, respectively. B and its children run at guardian G_1, and the other actions run at G_2. $B.1$ runs first, creating $B.1.1$, which writes an object X at G_2. Sometime after $B.1.1$ is created, $B.1$ aborts. Possible reasons could be communication problems or problems decoding the reply message for $B.1.1$. As a result of $B.1$'s abort, B creates $B.2$, (so $B.2$ knows that $B.1$ aborted). Then $B.2$ creates $B.2.1$, which requests a write lock for X. $B.2.1$ can know, based on information from its parent, that it should be able to get the lock at X.

3.2.3 Knowledge that an Action Effectively Never Accessed an Object

In the previous example it was possible that $B.2.1$ actually knew, in the usual sense of the term, that $B.1.1$ was effectively aborted. Our third example illustrates the case where an action may not know whether some other action accessed an object

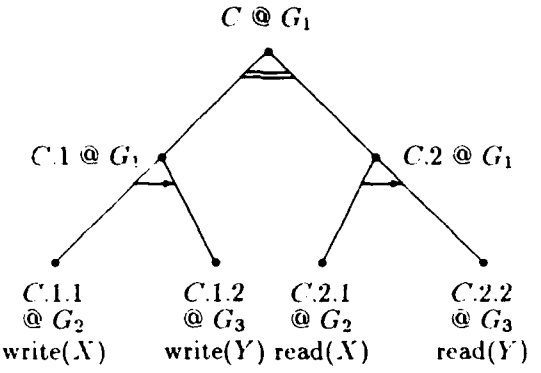


Figure 3.4: $C.2.2$ knows that $C.1.2$ effectively never accessed Y

and then was effectively aborted, or just never accessed the object, but it does know that one or the other is true.

Figure 3.4 shows an action C with two local children, $C.1$ and $C.2$, at guardian G_1. Each child has two children that run at guardians G_2 and G_3. $C.1$ and $C.2$ are concurrent siblings, and each of their sets of children are sequential siblings. The lowest level actions access object X at G_2 and Y at G_3. All programs follow the convention that X is always read or written before Y. The actions in the picture obey this convention by causing $C.1.1$ to run sequentially before $C.1.2$ and $C.2.1$ to run sequentially before $C.2.2$. The execution proceeds as follows. $C.1.1$ runs, writes X, and commits to $C.1$. Then $C.1.2$ runs and writes Y. In the meantime, $C.2$ starts running. At some point $C.1$ aborts and notice of $C.1$'s abort is sent to G_2, perhaps as a courtesy lock propagation message. $C.2.1$ then starts running and is able to read X since the information about $C.1$'s abort allows $C.1.1$'s lock to be released. At this point $C.2.1$ learns of $C.1.1$'s effective non-access at X. For future reference, notice that G_3 has not yet learned of $C.1$'s abort, so $C.1.2$ still holds a write lock on Y. Now $C.2.1$ commits up to $C.2$, and $C.2.2$ starts running and tries to read Y. $C.2.2$ expects to be able to get the lock on Y since it learns of $C.1.1$'s effective non-access at X from $C.2.1$, and this implies $C.1.2$'s effective non-access at Y.

3.3 A Simplified Protocol

This section describes a simplified version of the proposed commit protocol. Though obviously inefficient, it does contain the key ideas that form the basis for the optimized protocol. By considering this protocol first, we can argue the correctness of these ideas while ignoring the many optimization details.

The protocol we present first is not the simplest imaginable. A trivial protocol could just maintain sets of commit and abort information at guardians and send

all information on every message. Such a protocol is clearly correct—and clearly inefficient. A slightly less trivial protocol could reduce commit information sent in messages without employing any inference mechanisms. The protocol we describe incorporates the inference mechanisms.

The protocol is presented by first describing the data structures that must be maintained at each guardian and then describing how these data structures change as the states of actions change. We also describe the information that must be added to messages when guardians communicate.

3.3.1 Notation and Conventions

In the following description, we use a shorthand to refer to particular relatives of an action:

- $cpa(A)$ is the set of actions that are children of proper ancestors of action A. Similarly, $ca(A)$ is the set of children of ancestors of A (which includes children of A also).

- $top(T)$ is used to mean both the topaction of action T, if T is a single action, or the set of topactions of actions in T if T is a set of actions. The usage should be clear from context.

- $lca(A, B)$ is the action that is the least common ancestor of actions A and B.

In describing the protocol, we consider all topactions to be concurrent children of a mythical action called T_0. Thus a topaction is the "child of a proper ancestor" of every other action (except T_0)—the ancestor is T_0. This makes our descriptions simpler and more uniform.

Note that for the purposes of the protocol, nested topactions are treated as non-nested topactions.

3.3.2 Data Structures

Every guardian G maintains two sets of action identifiers, called $G.committed$ and $G.aborted$. When the guardian is created both sets are initially empty.

Every message M that contains information about the creation or completion of an action also contains two additional pieces of information, which we designate $M.committed$ and $M.aborted$. Such messages include call and reply messages for remote procedure calls, lock propagation messages, and messages used to carry out two-phase commit.

3.3.3 Propagating Commit and Abort Information

Now we describe the information that must be recorded and sent in messages as actions change states. For completing actions, the information indicated below must be recorded before any locks held by the action are released or propagated and before any later sequential actions are created. For remote actions being created, information must be recorded by the time the action or one of its descendants makes a lock request at the remote site. The chart in Figure 3.5 summarizes the discussion below.

Local Actions. A local action is an action that runs at the same guardian as its parent. The creation of a local action requires no special activity.

When a local action commits at guardian G, its action identifier is inserted into $G.committed$ if the action has concurrent siblings.

When a local action aborts at guardian G, its action identifier is inserted into $G.aborted$.

Remote Actions. Remote subactions are created by sending RPC call messages and are completed via RPC reply messages. Whenever a guardian sends an RPC message, it includes some of its commit and abort information in the message, as described below. The receiving guardian merges the aborted and committed sets in the message with its own aborted and committed sets, respectively.

When a guardian G creates a remote subaction A by sending an RPC call message M to another guardian G', G sets $M.aborted$ to $G.aborted$ and it sets $M.committed$ to contain $G.committed \cap cpa(A)$. That is, G sends its entire aborted set to G'. It filters its committed set to send only the action identifiers that will be useful to G'—those of actions that are children of proper ancestors of the action being created. We will argue in Section 3.5 why this part of the committed set suffices.

When a remote subaction A commits at guardian G, if A has concurrent siblings, then G adds A to $G.committed$. In the RPC reply message, G includes its entire aborted set $G.aborted$, and $G.committed \cap cpa(A)$.

When a remote subaction A aborts at guardian G (either voluntarily, or because the transaction manager at the guardian decided to abort it), G inserts A into $G.aborted$. In the RPC reply message M, G sets $M.committed$ to be empty and $M.aborted$ to contain just the action identifier for A.

Other Messages. The messages that concern us, other than RPC messages, are lock propagation messages and messages sent as part of two-phase commit. Whenever a guardian G sends a message M to a guardian G' saying that an action A committed up to some other action A', G sets $M.aborted$ to be $G.aborted$ and $M.committed$ to be $G.committed \cap ca(A')$. Upon receiving the message, G' unions the sets in the message with its own sets.

	Local (at guardian G)	Remote (G_1 sends M to G_2)
Create A		$M.aborted \leftarrow G_1.aborted$ $M.committed \leftarrow G_1.committed \cap cpa(A)$
Commit A	if $concurrent(A)$ then $insert(G.committed, A)$	if $concurrent(A)$ then $insert(G_1.committed, A)$ $M.aborted \leftarrow G_1.aborted$ $M.committed \leftarrow G_1.committed \cap cpa(A)$
Abort A	$insert(G.aborted, A)$	$insert(G_1.aborted, A)$ $M.committed = \emptyset$ $M.aborted \leftarrow \{A\}$
"A committed up to A'"		$M.aborted \leftarrow G_1.aborted$ $M.committed \leftarrow G_1.committed \cap ca(A')$
"A aborted"		$M.committed \leftarrow \emptyset$ $M.aborted \leftarrow \{A\}$

Figure 3.5: Simplified protocol—information to be recorded when starting and completing actions and when sending messages. In all remote cases G_2 merges $M.aborted$ into $G_2.aborted$ and $M.committed$ into $G_2.committed$.

Whenever a guardian G sends a message M to a guardian G' indicating that an action A aborted, G sets $M.committed$ to be empty and $M.aborted$ to contain the action identifier for A. G' unions the sets in the message with its own sets.

Crashes. Every time a guardian prepares for two-phase commit it must write $G.aborted$ and $top(G.committed)$ to stable storage. The topaction being committed could depend upon this information and thus it must survive crashes.

3.4 Lock Propagation

The committed and aborted sets at a guardian are used to propagate locks when an action A requests a lock on an object for which another action B holds a conflicting lock. Let C be the least common ancestor of A and B. (If A and B are descendants of different topactions then their least common ancestor is T_0. Recall that every topaction is a child of T_0.) We are interested in determining either that B has committed up to C, in which case B's lock can be propagated to C, or that an ancestor of B has aborted, in which case B's lock can be released. The following rules determine B's status based on the committed and aborted sets at the guardian where the lock request occurs. They are applied in order. Figure 3.6 illustrates the

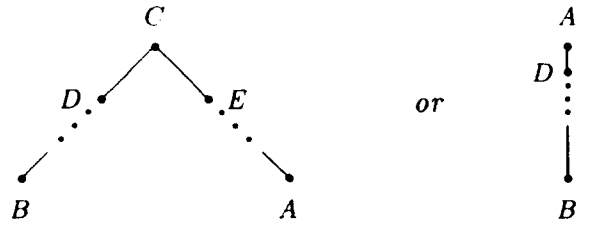


Figure 3.6: Lock propagation

relationships among the actions.

1. If any ancestor of B below C is in the aborted set, then B aborted.

2. Otherwise, if an ancestor of A is a later sequential sibling of an ancestor of B, or A is an ancestor of B, then B is committed up to C. (If A is an ancestor of B then $A = C$.)

3. Otherwise, if A and B are descendants of concurrent siblings, let D be the child of C that is an ancestor of B. If D is in the committed set, then B is committed up to C.

4. Otherwise, if an ancestor of A is a prior sequential sibling of an ancestor of B, then A must be an orphan and should not be permitted to continue.

5. Otherwise, we cannot locally determine whether B has committed up to C.

We are implicitly using two kinds of inferences in these rules. We use *sequential inference* in the second rule to determine that D must have completed. By the fact that A is running, we know that A's ancestor that is a child of C was created. And for A's ancestor to be a sequential sibling of D, it cannot have been created until D completed.

In both the second and third rules we use *descendant inference* to determine the completion status of B. We know that D has completed, either by sequential inference or by the fact that it is in the committed set. A simple inductive argument shows that if D completed then every descendant of D either committed up to C or has an ancestor that aborted (and thus is an orphan). We will argue in the next section that the aborted set contains enough information to make these inferences correctly.

In the fourth rule we use sequential inference to determine that A's ancestor E that is a child of C is completed. Since E's completion implies that A must be orphaned or committed up to E, and A is still running (requesting a lock), we know that A must be an orphan. We can supply this information to the system, which can

then destroy A. There is no reason to grant the lock to A since it will eventually be destroyed.

3.5 Correctness Arguments

Now we want to argue informally for the correctness of the simplified protocol. We want to prove that the protocol supports the eager diffusion semantics. In the process of doing so we will have to argue that the inferences made by the protocol are correct.

To begin with, imagine modifying the simplified protocol by replacing the inference mechanisms with explicit information:

1. All actions are added to the committed sets when they commit—not just those with concurrent siblings.

2. For a remote action A being created or committing, the committed sets sent in the RPC message include all proper descendants of proper ancestors of A that are locally visible to A according to the committed set, not just the children of proper ancestors.

3. In a lock propagation message saying that A committed up to A', the committed set in the message includes all proper descendants of ancestors of A' that are locally visible to A' according to the committed set, not just the children of ancestors.

In the first case we are removing the sequential inference mechanism, and in the second and third cases, the descendant inference mechanism. First we argue that this simpler protocol guarantees the correctness property. Then we argue that it is correct to relax each of the conditions above, replacing them with the corresponding inference mechanism.

3.5.1 Correctness of Simpler Protocol

We begin by arguing that the simpler protocol supports the eager diffusion semantics. Suppose A is an action requesting a lock for an object X at guardian G, and B is an action that could hold a conflicting lock on X. Further suppose that A knows that it should be able to obtain the lock on X. We want to argue that G will have enough information to grant the lock to A.

Recall that in Section 3.1 we listed three conditions under which A would be able to obtain the lock on X when B is a potentially conflicting lock holder:

C1: B is committed up to its least common ancestor with A, or

C2: B has an aborted ancestor, or

C3: B never obtained the lock on X.

For A to know that the lock on X should be available it must know the disjunction of these conditions. We will consider the direct and indirect sources of information available to A that allow A to know this disjunction. We use an informal, inductive style of argument to prove our claim that G will also have enough information to know the disjunction of the conditions in each case.

First we consider the direct sources of information available to A.

1. Children. If B is a committed (or aborted) child of A, then A knows condition C1 (or condition C2) and thus knows the disjunction of the conditions. If B is a local child then its commit (or abort) is recorded directly at G when it occurs. If B is a remote child then it will be added to G's committed (or aborted) set when G receives the corresponding RPC reply message. In either case, G also knows C1 (or C2) and thus knows the disjunction of the conditions.

2. Objects. If B is visible to A at a local object Y, then A knows C1. In this case G will also know C1 because, as B committed up to $lca(A, B)$ at object Y, G recorded the commits of each ancestor of B in its committed set.

   If A knows, by accessing an object Y, that an action D effectively never accessed Y, and if D's effective non-access at Y implies B's effective non-access at X, then A knows the disjunction of C2 and C3. Now if B never did access X, then G knows C3. If B did access X but was effectively aborted, the rules on page 33 tell us that the same ancestor that caused D to be effectively aborted has caused B to be effectively aborted. Thus, that ancestor was added to G's aborted set before A accessed Y, so G knows C2.

Now we consider indirect sources of information available to A. If A knows of the commit of an action C, or if C is A's parent, then A knows everything that C knows. We assume, inductively, that C's guardian knows as much as C knows. So clearly, when C and A run at the same guardian, G knows what C knows. Assuming that C runs at a guardian other than G, we consider cases for C's knowledge, showing that G will know as much as A knows.

1. C knows that B committed up to its least common ancestor with C.

   (a) If C is A's parent, then $lca(A, B) = lca(C, B)$. So A knows C1. The RPC call message for A contains in its committed set all actions that are visible to C at C's guardian. This includes all ancestors of B below $lca(A, B)$, so G will know C1 also.

   (b) If C is a committed child of A, then either $lca(C, B) = C$ or $lca(A, B) = lca(C, B)$. In either case, A knows C1. The RPC reply message for C contains in its committed set all actions that are visible to C at C's guardian.

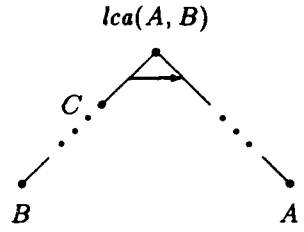


Figure 3.7: Action tree for descendant and sequential inference correctness arguments.

This includes all ancestors of B below $lca(C, B)$, as well as C itself, so G will know C1 also.

(c) If C is an action visible to A at a local object Y, then either $lca(A, B) = lca(C, B)$ or $lca(C, B)$ is an ancestor of C below $lca(A, C)$. In either case, A knows C1. The message that allows C's lock on Y to be propagated to A contains in its committed set all actions that are visible to C at C's guardian. This includes all ancestors of B below $lca(C, B)$ as well as all ancestors of C below $lca(C, A)$. Thus G will know C1 also.

2. C knows that an ancestor of B aborted. Thus A knows C2. The message associated with C that arrives at G will contain C's guardian's entire aborted set. (The type of the message will be RPC call if C is A's parent, or RPC reply if C is A's child, or lock propagation if C is visible to A at an object). Thus G will also know C2.

3. C knows that an action D effectively never accessed an object Y, and A knows that D's effective non-access at Y implies B's effective non-access at X. So A knows the disjunction of C2 and C3. If B has never accessed X, then G knows C3. If B has accessed X, then the rules on page 33 guarantee that an ancestor of B will be aborted and it will be the same ancestor that has caused D to be effectively aborted. So G will know C2 when it receives the message associated with C.

This concludes the correctness arguments for the simpler protocol.

3.5.2 Correctness of Descendant Inferences

Suppose that action A is requesting a lock for which action B holds a conflicting lock. Now we observe that in order to determine whether B's lock may be given to A we do not actually need explicit commit information for all actions visible to A—only for those actions that are children of proper ancestors of A. Let C be the child of $lca(A, B)$ that is an ancestor of B (see Figure 3.7). If C is not in the committed set

and no ancestor of B is in the aborted set, then A cannot obtain a lock from B since B is not visible to A and is not aborted. If C is in the committed set, then we can infer whether or not B is committed up to C.

Suppose C is in the committed set. By a simple inductive argument, B must either be committed up to C or have an aborted ancestor that is a proper descendant of C. We claim that B committed up to C if and only if no ancestor of B below C is in the aborted set. Another way to say this is that an ancestor of B below C aborted if and only if B has some ancestor below C in the aborted set. Since the aborted set contains only actions that have aborted, if an ancestor of B below C is in the aborted set, then that ancestor has aborted. So we must show that if an ancestor of B below C is aborted then B will have some ancestor below C in the aborted set. Suppose that none of B's proper ancestors aborts. Then B itself must have aborted and this will be known to its parent, which will pass the information up its ancestor chain to C as the actions on the chain commit up to C. Thus, any site that has information about C's commit will also have information about B's abort. If one of B's proper ancestors below C does abort then the same argument applies for the highest aborted ancestor of B below C. So one of B's ancestors below C will be in the aborted set at A's guardian.

3.5.3 Correctness of Sequential Inferences

Again, suppose that action A is requesting a lock for which action B holds a conflicting lock. In order to determine whether B's lock may be given to A, we do not actually need explicit commit information for all children of proper ancestors of A—only for those that have concurrent siblings.

Let C be the child of $lca(A, B)$ that is an ancestor of B (as in Figure 3.7). If C does not run concurrently with any of its siblings, then any sibling that runs once C has been created can infer that C must have completed. In particular, A's ancestor that is a sibling of C must be running given that A is, and we know C was created given that B was, so we conclude that C must have completed. Furthermore, if C had aborted, its parent would have known of the abort at the time it created A's ancestor and would have conveyed that information to A's ancestor, which would have eventually conveyed it down the ancestor chain to A. Thus if C was aborted, A's guardian would have C in its aborted set. So if C is not in the aborted set at A's guardian, then C must have committed and we can proceed to figure out whether B committed up to C. If C is in the aborted set, then B has an aborted ancestor and its lock may be given to A.

3.6 Examples Revisited

We now return to the examples described in Section 3.2, explaining how our protocol guarantees the eager diffusion semantics in each case.

3.6.1 Knowledge of a Commit

The example in Figure 3.2 (page 35) illustrated knowledge of a sequential commit. This example is similar to the problems encountered by [Greif *et al.* 1987] discussed in Chapter 1. In a system using a lazy diffusion protocol, it is possible that when $A.2$ requests the lock, G_2 has not yet been notified that $A.1$ committed up to A and has no means of inferring this. So the lock request would cause lock propagation queries to be sent to G_1. Using our protocol, G_2 will be able to infer that $A.1$ had committed up to A from the fact that $A.1$ was sequentially before $A.2$ and no ancestor of it below A is in the aborted set.

Notice that in this example, there is actually no extra information being sent that allows G_2 to propagate the lock. Rather it is the lack of information about an abort, in conjunction with the inference mechanisms, that allows G_2 to know that $A.1$ committed.

3.6.2 Knowledge of an Abort

The example in Figure 3.3 (page 35) illustrated the case of an action knowing about another action's effective abort. It is possible under lazy diffusion that news of $B.1$'s abort would not reach G_2 by the time $B.2.1$ requests the lock. However, with our protocol, $B.1$'s identifier is added to G_1's aborted set when $B.1$ aborts. Then G_1's aborted set is sent to G_2 along with the call message for $B.2.1$, so G_2 can use descendant inference to release $B.1.1$'s lock.

3.6.3 Knowledge that an Action Effectively Never Accessed an Object

Figure 3.4 (page 36) illustrated knowledge about an action's effective non-access at an object. Our protocol will ensure that the notice of $C.1$'s abort arrives at G_3 along with the call message that creates $C.2.2$. When information of $C.1$'s abort arrives at G_2 and $C.1.1$'s lock is released, $C.1$ is added to G_2's aborted set. When $C.2.1$ commits to $C.2$, G_2's aborted set is contained in the reply message and is unioned into G_1's aborted set. Then when $C.2.1$ is created, G_1's aborted set is included in the call message to G_3. Thus G_3 knows of $C.1$'s abort.

3.6.4 Knowledge of a Concurrent Commit

We give one additional example to illustrate how information in the committed set is used to propagate locks.

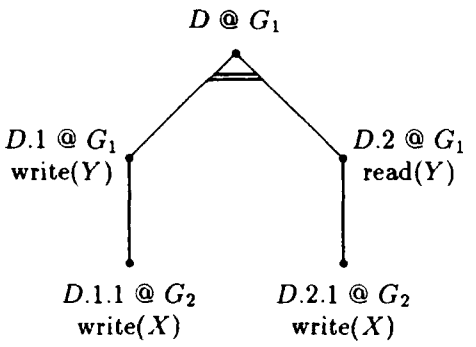


Figure 3.8: $D.2.1$ knows that $D.1.1$ (a concurrent action) committed up to D.

Figure 3.8 shows an action D at guardian G_1, with two local children $D.1$ and $D.2$ that are concurrent siblings. Each child of D has a remote child that runs at G_2. There are two objects, Y at G_1, and X at G_2. The execution proceeds as follows. D creates $D.1$ and $D.2$. $D.1$ creates $D.1.1$, which writes X and commits up to $D.1$. Then $D.1$ writes Y and commits up to D. So $D.1$ may have recorded at Y that $D.1.1$ committed. Now $D.2$ reads Y and then creates $D.2.1$, which will try to write X. Since $D.2$ could have learned through Y of $D.1.1$'s commit and conveyed this information to $D.2.1$, $D.2.1$ could expect to get the lock at X.

Using our protocol, the information will be available at G_2 to propagate $D.1.1$'s lock to $D.2.1$. When $D.1$ commits to D it will be added to the committed set at G_1 since it has a concurrent sibling. Then when $D.2$ sends the call message to create $D.2.1$, it will include $D.1$ in the committed set since $D.1$ is a child of a proper ancestor of $D.2$. G_2 can then infer, from the fact that $D.1$ is committed and no ancestor of $D.1.1$ is aborted, that $D.1.1$ must have committed up to D, which is its least common ancestor with $D.2.1$.

Chapter 4

Optimizations and Efficiency Issues

In this chapter we present optimizations and discuss other issues involved in obtaining an efficient implementation of the previous chapter's protocol. We also consider how our ideas may be adapted to improve the efficiency of a protocol that supports only the lazy diffusion semantics.

We begin in Section 4.1 with optimizations. First we present a series of simple optimizations for reducing the amount of information maintained by the protocol and for organizing the data structures efficiently. These optimizations are clearly useful and should improve the efficiency of any implementation. Then we consider some additional optimizations that involve more questionable tradeoffs and discuss their merits. As we present each optimization we argue informally for its correctness.

The most glaring efficiency problem in the simplified protocol is that aborted and committed sets grow without bound—the protocol does not describe how to discard old information from the sets. It turns out that this is a fairly hard problem to solve (and not a new one in distributed systems). We examine this problem in Section 4.2, separately from the other optimizations, and propose a possible solution. Our proposal is based on a solution designed for garbage collecting orphan detection information in Argus. A good solution to this problem is required if the protocol is ever to be practical.

In Section 4.3 we describe the interactions of the protocol with orphan detection. We show how the data structures required for abort orphan detection may be combined with the data structures in the protocol to yield a more efficient implementation. We also explain why crash orphan detection is crucial to the ability of the protocol to support the eager diffusion semantics.

In Section 4.4 we address the issue of early release of read locks during two-phase commit. This is not so much an efficiency issue for our protocol as it is for the rest of the action system. As we mentioned in Chapter 2, a well-known optimization for two-phase commit is to allow read locks to be released during the first phase. We ignored that optimization in the previous chapter in order to keep the explanation of the basic protocol as simple as possible. Since we realize that the optimization is an important one, we consider its effect on our protocol in this chapter.

Finally, in Section 4.5, we consider how ideas from the protocol may help to improve the efficiency of nested action commits and aborts in Argus.

4.1 Optimizations

There are a number of optimizations that we can apply to the simplified protocol that will improve the expected efficiency of an implementation. They fall into three categories:

1. Observations about the behavior of the protocol that allow us to reduce the sizes of the committed and aborted sets—with obvious efficiency gains.

2. Optimizations in organizing and operating on data.

3. Optimizations to reduce the sizes of the committed and aborted sets where a non-negligible tradeoff in communication and computation time is involved.

We proceed to describe each of these optimizations and indicate why they preserve the correctness of the protocol.

4.1.1 Simple Reductions in Set Sizes

The first observation is that there is no need to add an action's identifier to a committed or aborted set if the action has no committed remote descendants. The only reason to propagate the outcome of an action A is that A may have descendants at other sites that had obtained locks and then committed up to A. If another action B observes A to be completed and then visits a site where a descendant of A ran, B must observe A's descendant to be completed up through A. If the descendant had committed up to A, then we can guarantee that the descendant's site receives the information by adding A to the committed or aborted set at A's guardian, as appropriate. If the descendant or one of its ancestors below A had aborted, this information would be recorded in the aborted set at A's guardian already. The question is how the system can determine when A does *not* have committed remote descendants. We can safely assume it to be true if A never made a successful remote call—in other words, either A never made a remote call, or, if it tried to, no call message was ever sent. The system could remember even more information than this and convey it back to A's guardian on RPC reply messages for remote descendants if it seems useful and not too costly. For example, if A makes a remote call that aborts, the called guardian may be able to inform A's guardian when the aborted action had no committed descendants.

A second observation that reduces the sizes of aborted sets is that when an action

is inserted into an aborted set, all of its descendants in the set may be removed.[1] This is permissible because the abort of an action results in the effective aborts of all of its descendants, and any changes they made will eventually be undone. We must be careful, however; the optimization has an effect on the locking rules in some cases where the lock requestor is an orphan.

Recall that the locking rules require us to check for aborts of ancestors of the lock holder only up to its least common ancestor with the lock requestor in the process of deciding whether to undo the effects of the lock holder. To see how an orphaned lock requestor can affect the behavior of the system, suppose A is a lock requestor, B holds the conflicting lock, and C is B's ancestor that is a child of $lca(A, B)$ (as in Figure 3.7 on page 43). If A is an orphan due to the abort of an ancestor of $lca(A, C)$, and hence C is also an orphan, then it is possible that the only ancestor of B in the aborted set is a proper ancestor of C. The locking rules would cause us to infer that B was committed up to C in this case, which may not be true (*i.e.*, if one of B's ancestors below C happened to get removed from the aborted set when the ancestor above C was added). Orphans will be detected before they do any harm. However, if we are not convinced that it is reasonable to allow the orphan to see the incorrect inference, then it is possible to use the information in the aborted set to cause the system to detect A as an orphan and destroy it before it executes any further.

A third observation is that once all sibling actions in a concurrent group of actions complete (where the concurrent group may consist of a single action), the entire group may be removed from the committed set. This is valid because any later action encountering a lock held by an action in the concurrent group will be able to infer the completion status of the concurrent action by the sequential inference mechanism. If this optimization is applied at each opportunity, then when an action commits, none of its proper descendants will remain in the committed set. Similarly, when an action aborts and its identifier is inserted into the aborted set, all of its descendants may be removed from the committed set.

A final observation is that we can use the fact that a topaction committed to encode the aborts of its descendants. Once a guardian has inserted a topaction's identifier into its committed set, it may remove all descendants of the topaction from the aborted set. This optimization relies on the special nature of topaction commit. In the process of committing a topaction we propagate to the topaction all locks held by its descendants that are committed up to the top. Once the topaction is in a committed set, each of its proper descendants that still holds locks and that is not prepared for two-phase commit must have an aborted ancestor.[2] Furthermore, the guardians where the locks are held could not have been participants in the topaction's two-phase

[1] This optimization has been used in the Argus abort orphan detection algorithm to reduce the size of its *done* data structure.

[2] If we assume that locks are propagated from subactions to topactions during phase one of two-phase commit then we need not check for prepared proper descendants.

commit because otherwise they would have been notified of the descendant's abort, which would have caused them to release the locks. In some sense, these descendants are orphans—they are actions that have aborted ancestors—and we might hope that an orphan detection algorithm would eliminate them before they could get in the way. However, these orphans are passive in that they are not currently running and so cannot see inconsistent states. The Argus abort orphan detection algorithm and similar algorithms that we know of do not promise to destroy these types of orphans.[3] We must deal with them ourselves if the protocol is to support eager diffusion.

If we choose to apply this final optimization we must change the lock propagation rules to account for it. We use the rules described in Section 3.4 in the case where the lock requestor and lock holder are descendants of the same topaction. In the case where they are descendants of different topactions, we use the rules below (for A the lock requestor, B the lock holder, $C = top(B)$). Note that these rules rely on the assumption that once C is in the committed set, all descendants of C that committed to the top will have had their locks propagated to C.

- If C is in the committed set, then: if $B = C$, then B is committed, and otherwise B is aborted.
- If any ancestor of B is in the aborted set, then B is aborted. If no ancestor of B is in the aborted set, then we cannot locally determine whether B has committed through C.

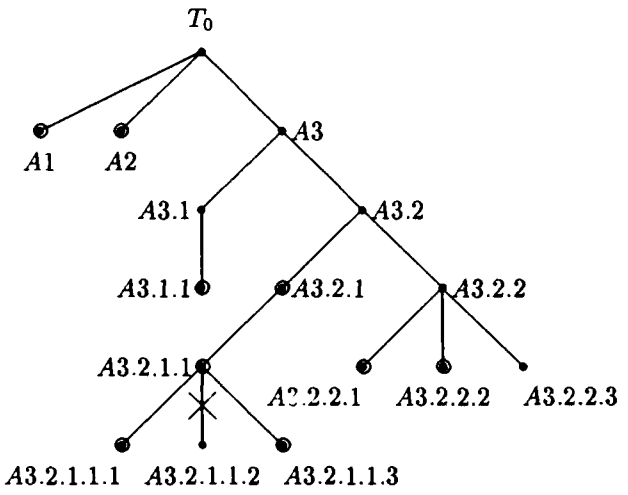
After applying the above optimizations we can characterize the contents of the committed and aborted sets at a guardian as follows:

- Aborted sets contain identifiers of actions that aborted and may have remote descendants that are committed up to them. For any action in the aborted set, no descendant of the action is also in the set.
- Committed sets contain identifiers of actions that committed, have concurrent siblings, may have committed remote descendants, and whose proper ancestors are not in the committed or aborted sets.

4.1.2 Organization of Data Structures

A common operation on committed sets is to extract children of proper ancestors of a given action from the set and place it in a message. The other operations on committed sets are membership test, insertion, and deletion. We can take advantage

[3] Actually, the Argus algorithm will catch this kind of orphan if it may have been active when its ancestor aborted. If so, the Argus algorithm will carry around information about the aborted ancestor that will allow it to detect the possibly running orphaned action.



Action	Committed Children
T_0	A1, A2
A3.1	A3.1.1
A3.2	A3.2.1
A3.2.1	A3.2.1.1
A3.2.2	A3.2.2.1, A3.2.2.2
A3.2.1.1	A3.2.1.1.1, A3.2.1.1.3

Figure 4.1: Representation of the committed set.

of the hierarchical structure of the action identifiers in these sets to allow for efficient extraction of children of ancestors and for efficient insertion and deletion of action identifiers. We propose to represent the committed set as a table of sets of sibling actions, where the key for a sibling set is the parent's action identifier. An identifier for the "mythical" T_0 action serves as the key for the set of topaction identifiers. We can also take advantage of the hierarchical structure of action identifiers to allow for efficient access to ali the ancestor keys when extracting the children of ancestors of an action.

Figure 4.1 shows an action tree where the circled actions are in the committed set and the action with an 'X' through its branch is in the aborted set. The corresponding representation of the committed set is shown next to the action tree. To extract children of proper ancestors of action A3.2.2.3, we simply look up A3.2.2, A3.2, A3, and T_0 in the table.

When composing a message containing a committed set, we extract the appropriate sibling sets, preserving the table structure in the message. The recipient of the message can easily merge the committed set in the message with its own by simply unioning sibling sets at matching keys and inserting new entries into its table for keys not already there.

The expected size of committed sets being sent in messages becomes particularly clear given this structure. We can expect each sibling set (other than the top-level one) to contain only committed actions that may have committed remote descendants and that are in the last concurrent group not known to have finished. In other words

Action	Committed Children	Completion Flag
T_0	$A1$, $A2$	?
$A1$		committed
$A2$		committed
$A3$		?
$A3.1$	$A3.1.1$	?
$A3.2$	$A3.2.1$	?
$A3.1.1$		committed
$A3.2.1$	$A3.2.1.1$	committed
$A3.2.2$	$A3.2.2.1$, $A3.2.2.2$	?
$A3.2.1.1$	$A3.2.1.1.1$, $A3.2.1.1.3$	committed
$A3.2.2.1$		committed
$A3.2.2.2$		committed
$A3.2.2.3$		?
$A3.2.1.1.1$		committed
$A3.2.1.1.2$		aborted
$A3.2.1.1.3$		committed

Figure 4.2: Using completion flags as redundant encodings of committed and aborted sets.

we can expect these sets to be quite small. Of course, the sibling set mapped to T_0 contains all committed topactions. In a properly functioning system, we would hope that this set is large, because work is done by committing topactions. This is where we run into possible efficiency problems.

While we do not need to perform any complicated extraction operations on aborted sets, we can also take advantage of the structure of action identifiers to organize aborted sets to allow for efficient insertion, deletion, and lookup operations.

Another data structure optimization for both the committed and aborted sets is to keep redundant information about the contents of the sets in a table keyed on action identifiers (it could be the same table used for the committed set). Figure 4.2 shows how such a table would look for the action tree in Figure 4.1. The table maps each action identifier to a *completion flag* indicating whether the action is aborted, committed, or still running. Each time an action is inserted into one of the sets, its flag is set appropriately. This optimization is useful if the table lookup for an action identifier can be made more efficient than a search in the sets. This would be true, for example, in the suggested representation for the committed set, where an action identifier lookup for the parent would be required to find the child.

It may not be practical to require that completion flags are always set when the committed or aborted sets are updated. For example, we may not want to delay processing of a call or reply when a message arrives at a guardian containing committed and aborted sets. In this case we can consider the flags as an optional optimization—a cache—that is checked before a lookup in the sets is attempted. The locking rules

can easily be modified to take the completion flags into account.

We can carry the idea of completion flags even further by recording even more information. Rather than just recording that an action has committed, we can record the ancestor to which it is known to be committed. This eliminates some redundant inferences during lock propagation. The completion flags can also be used by a background process to propagate all locks held by an action based on the completion statuses that are inferred in the course of releasing a particular lock.

4.1.3 Further Reductions in Set Sizes

To some extent, we can make a tradeoff between the sizes of the committed and aborted sets and the amount of delay and communication we require when actions commit and abort. It is difficult to judge, outside the context of a particular system and set of applications, whether the cost to reduce the set sizes is tolerable.

The Committed Set. As the protocol is currently defined, a participant in a two-phase commit protocol adds the topaction to its committed set when it learns of the decision to commit the action during phase two. Once the action is in the committed set, it remains there and will continue to be sent out in messages originating at the guardian (recall that a topaction is the child of a proper ancestor of every action). We would like to know whether there is any point at which we can safely remove the topaction from the committed set.

First, we remark that if the last optimization suggested in Section 4.1.1 has been applied (where topactions in the committed sets encode aborts of descendants), then we cannot hope to remove an action from the committed set unless we are sure that all of its remaining orphaned descendants will be detected by the abort orphan detection mechanism. In Argus, this will be the case if all descendants of the topaction in the aborted set that would be removed by that optimization are also known by the orphan detection algorithm to have been aborted.

Given the above caveat, a guardian can safely remove a topaction from its committed set once it knows that all participants in the action's two-phase commit have been informed of the commit decision. The coordinator of the two-phase commit will know this at the end of phase two when it receives acknowledgments from the participants. Thus, it may remove the topaction from its committed set after phase two completes, or it may avoid adding the topaction to its committed set entirely if it waits until the end of phase two to release the topaction's locks (recall that it may release the locks at the end of phase one, once it knows the decision).

Currently, there is no way to detect at a participant that all other participants have been informed of the commit decision. A possible solution is to add a quick third phase to the commit of a topaction in which the coordinator attempts to notify participants that the two-phase commit is complete. Participants may remove the

topaction from their committed sets upon receiving this phase three message, or they may delay releasing locks held by the topaction until they receive the message and thereby avoid adding the topaction's identifier to their committed sets at all. A participant may, at any time, decide not to wait for the phase three message and just add the topaction's identifier to its committed set and release the topaction's locks. In this way we are making the optimization optional at each participant and not requiring each to be delayed indefinitely after receiving a phase two message. We can further optimize the case of a two-phase commit involving only one participant other than the coordinator. If the participant knows that it is the only one, then it knows that all participants are informed of the commit decision when it receives the phase two commit message from the coordinator, eliminating the need for a third phase.

The Aborted Set. We can use a similar idea to avoid adding an action's identifier to the aborted set when the action aborts. The idea is to carry out an exchange of messages when an action aborts, attempting to notify its committed descendants' guardians of the abort. This optimization would be useful only if none of the aborting action's descendants are already in the aborted set.

Suppose action A is aborting at guardian G. For every descendant B of A that has committed up to A (this information is known at A's guardian), G sends a message to B's guardian indicating that A has aborted. B's guardian acknowledges this message. Then, after receiving acknowledgments from each such B, G sends a second message to each one indicating that all have acknowledged the first message. Upon receiving this message, each of the descendants' guardians may release any locks held on behalf of A or its descendants without adding A to its aborted set. As before, each of the guardians may time-out in waiting for the messages and simply add A to their aborted set before releasing its locks. The case where there is only one other guardian involved besides G may be further optimized to use only two phases, as above.

4.2 Garbage Collection of Top-Level Information

The most significant efficiency problem with the protocol as it stands is that aborted and committed sets grow without bound. We suggested optimizations that can slow the growth, but this is still a problem for any long-lived system running many actions. The desirable property for the system—that it run and commit actions as often as possible—is exactly the cause of the efficiency problems with the protocol.

There are really two problems:

- Detecting when it is safe to remove an action identifier from a committed or aborted set, and
- Preventing a deleted action identifier from reappearing in the set from which it was removed.

Unlike some systems with distributed garbage collection problems we have the problem that we can effectively never discard information. The possibility of communication failures means that guardians holding locks on behalf of actions that have since been orphaned could be partitioned for indefinite periods of time from the guardians that know the completion status of the action. A system such as Argus does not even try to remember which guardians ran orphaned actions, so it is never possible to determine that all guardians that need to know of an action's abort have learned of it.

A similar type of garbage collection problem exists in the Argus abort orphan detection algorithm. The Argus designers have proposed two interesting solutions, one of which may be adapted to work for our protocol. However, the expected performance of the method in the case of orphan detection does not reflect directly on our protocol. The data structure that must be reduced in the abort orphan detection algorithm is a set of action identifiers of topactions and aborted subactions called *done*. An ancestor may replace its descendants in *done*, as in the aborted set in our protocol. Also, a completed topaction may replace its descendants in *done*, even if the topaction committed. Thus, *done* contains identifiers of aborted actions, and topactions that have aborted descendants. It is reasonable to argue that aborts happen infrequently enough that *done* will not grow too large too quickly. In our case, the committed set contains identifiers of potentially all committed actions, and for that reason we do hope that the set will grow quickly. The conclusion then is that the garbage collection method must certainly be tuned differently to be used in our protocol and it is not clear at this point whether it can be tuned well enough to be practical.

We mentioned that only one of the Argus solutions is suitable for our protocol. The problem with the other method is somewhat subtle and we think that its explanation provides some insight into the differences between the two protocols. We will describe both methods, indicating why one does not work and how to adapt the other for our purposes. The basic idea in both solutions is that although we cannot discard information, we can encode it. The important information that must remain available in the orphan detection algorithm, and in our protocol as well, is that particular topactions have finished.[4] Rather than remember that individual actions have finished, we place a time constraint—either logical or physical—on the actions and simply remember that all actions whose time constraint no longer holds must be finished. Thus, both methods work by limiting the lifetimes of actions in such a way that any guardian can detect when an action's lifetime has expired. Since the orphan detection information, as well as the information in committed and aborted sets, is used to determine the outcomes of actions, once we know that an action's lifetime is

[4]We can get away without knowing whether a topaction committed or aborted because any guardian that needs to know that the action committed will find out during two-phase commit. We made use of this property for the last optimization of Section 4.1.1.

over we can remove the action's identifier from the sets.

The method that causes problems is the one based on physical time constraints. The high-level reason is that we have control over the advance of logical clocks, while we do not have this control with physical clocks. There is a critical period (a "window of vulnerability") during two-phase commit, after participants have prepared but before they learn the outcome, where we must be able to avoid advancing the clock. We explain this in more detail below. First we describe the physical time limit scheme and explain why it does not work. Then we describe the logical time limit scheme and explain how to adapt it for our protocol.

4.2.1 Physical Time Limits

The first method limits lifetimes of actions by timing out individual topactions using physical clocks. It is described in [Walker 1984, Liskov *et al.* 1987c]. The technique requires that nodes have loosely synchronized clocks, *i.e.*, that there be some value ϵ that bounds the clock skew among the nodes [Lundelius 1984, Marzullo 1983].

Basic Method. When each topaction is created it is assigned a *deadline*, which is some time later than the current time of the clock at the node where the action is created. The deadline can be much larger than the local clock; the choice of setting depends upon the expected lifetime of the action. An action's deadline is included in its action identifier. It is inherited by all descendants of the action.

A guardian may consider an action's lifetime to be over once the clock at the guardian's node exceeds the action's deadline. Taking the clock skew into account, it will then be safe to remove the action from *done* as soon as the local clock is greater than the action's deadline plus ϵ. After this point the guardian can determine that the action has completed simply by comparing its deadline with the local clock. Waiting the additional ϵ guarantees that all other guardians can draw the same conclusion.

Problems in Adapting the Method. At first glance it seems that this scheme may essentially be adapted unchanged for eliminating information from committed and aborted sets in our protocol. Once a local action's deadline expires and the ϵ time period elapses, a guardian would assume that the action is aborted, if it has not heard otherwise, and release the action's locks. No action would be allowed to commit once its deadline expires. However, problems arise once an action prepares for two-phase commit. The action's deadline could expire after the action has prepared and while participants are waiting to hear of the outcome, but the action can no longer be aborted unilaterally by any guardian other then the coordinator. For concreteness, suppose action A is prepared for two-phase commit and action B runs at a guardian G that is a participant in A's two-phase commit; G could also be the coordinator. Now suppose that G has heard the outcome of the two-phase commit and has released

A's locks. So B could know A's outcome. Further suppose that A's deadline has expired, causing G to remove A from its sets. Now B visits another participant that has not yet learned the decision for A. B knows that it should be able to obtain any locks held by A, but the participant does not necessarily know it and there will be no information in the committed or aborted sets in the message for B that will help. We will see that in the method using logical clocks it is possible to ensure that A's "deadline" does not expire in the interval where the commit decision for A has been made but not all participants have been notified.

4.2.2 Logical Time Limits

The second method limits lifetimes of actions by timing out groups of actions using counters at guardians. The counters are, in effect, logical clocks [Lamport 1978]. The basic method is described in [Liskov *et al.* 1987c].

Basic Method. Each guardian maintains a stable *generation count*, which it increments periodically. When a topaction is created, the current generation count at the creating guardian is stored in the action's identifier; this is called the action's *generation*. Descendants of the topaction inherit its generation. The idea is that an action's lifetime expires when the guardian that created its topaction moves to a later generation.

Guardians enter their generation counts in a *generation map* at a logically centralized server. The server maps each guardian to a generation count and also maintains an associated *generation timestamp*. Generation maps with older information have earlier timestamps. The server provides operations to change the generation count of a guardian and to read the generation map. By *logically* centralized we mean that in a distributed system we would expect the server to be physically replicated to ensure high availability. (A scheme for replicating such a server is described in [Liskov 1987].)

Each guardian maintains a copy of the generation map and generation timestamp, which it obtains from the server. After a guardian advances its generation count, it waits for the completion of all of its topactions that belong to previous generations and then updates its entry in the generation map at the server. The call to the server will return a new generation map and generation timestamp.

Messages contain the generation timestamp of the sending guardian, along with all action completion information that was already being sent. The timestamp in the message is used by the receiving guardian to determine whether it must obtain more recent generation map information. If its own generation timestamp is older than that of the sender then it must communicate with the server to obtain information at least as recent as the sender's (in order that it "know" everything that the sender "knows").

The purpose of the generation counts is to allow the deletion of action completion information that was previously being sent in messages. As soon as an action's generation is older than the current generation for the guardian of its topaction, we know that the action is finished. Thus the completion of many actions is encoded in the generation map maintained at guardians, and the generation timestamp allows a guardian to detect when its map is out of date.

Adapting the Method. We use the generation map information to determine when lock holders have aborted. Let $G.gmap$ be the generation map for guardian G. Define $generation(A)$, for an action A, to be the action's generation, and $home(A)$ to be the guardian of A's topaction. An action A may be removed from $G.aborted$ or $G.committed$ once $generation(A) < G.gmap(home(A))$.

Timestamps in messages are used as follows. Each message M containing $M.aborted$ and $M.committed$ also has a timestamp, $M.ts$, which is the generation timestamp of the sending guardian. When the message is received at guardian G, if $M.ts > G.ts$, or if they are incomparable, then G calls the refresh operation of the server to obtain an up-to-date generation map. If $M.ts \leq G.ts$, then nothing special need be done since G's generation map is at least as current as the sender's. (If $M.ts < G.ts$ then G may be able to discard some action identifiers from $M.aborted$ and $M.committed$ based on its more recent generation map.) We can continue with normal processing while waiting for the results of a refresh operation until the point where an action attempts to get a lock and would fail. At that point we must have current information to decide what to do about the lock request.

The modified lock propagation rules of Section 4.1.1 must also change slightly to make use of generations. Suppose A wants to obtain a lock when B holds a conflicting lock and A and B are descendants of different topactions . Let $C = top(B)$. Before concluding that we cannot locally determine enough information about B, we must first check whether $generation(B) < G.gmap(home(C))$. If this is true, then B is effectively aborted. (C must have been removed from the committed or aborted set.)

Finally, the crucial change that permits this scheme to work is that a guardian may not update its generation count while any topaction with that generation is undergoing two-phase commit. If the guardian does want to update its generation count and no topaction is in the middle of phase two, then it may choose, as the coordinator, to abort all topactions in phase one, thus ending their two-phase commits. However, once the guardian has made the decision to commit a topaction, it is forced to wait to update its generation count until it receives acknowledgments from all participants that they have heard the outcome.

This last point can pose problems. The time to complete two-phase commit is unbounded, since participants may crash and partitions may prevent participants from communicating with the coordinator. A long delay in updating generation

counts could have serious consequences for the performance of our protocol. If a guardian cannot update its generation count but continues to run new topactions then the committed and aborted sets could grow to be very large. We can devise methods of dealing with this problem if it does not arise too frequently.

The basic cause of the problem is that lifetimes of all actions that run at a guardian are tied together in the generation map by the common name of the guardian. Starting a group of topactions in a new generation automatically implies that previously created topactions will be in an old generation. When a guardian encounters long delays in completing a two-phase commit, it might like to start new topactions in a different generation without forcing the generation of previous topactions—in particular, the one whose two-phase commit is running—to become old. The guardian cannot do this if it must enter a new generation count into the generation map with the same name it used for the current generation count. However, we could allow the guardian to "change its name" and enter a new generation count under a different name than the old one. In effect, we allow a guardian to have multiple, independent, "current generations" at a time. If extended delays in two-phase commit do not arise often, it will be sufficient for a guardian to maintain a small fixed set of names to cycle through.

The multiple-name scheme could work as follows. Every guardian has a circular list of names, with a designated *current name*. When a two-phase commit is delayed, the guardian switches its current name to the next name in the cycle and enters in the generation map a generation count for that name that is larger than the previous generation count for the name. It must be the case that all actions that previously started with that name and its old generation count have finished running. Then the guardian may start new topactions, inserting the generation count for its new name, as well as the name itself, into the topactions' identifiers. Once the delayed two-phase commit with the old name finishes, the guardian may increase the generation count for that name also, allowing the identifier for the action being committed under the old name to be garbage collected. This scheme will solve the problem, provided that every delayed two-phase commit completes by the time the current name of the guardian cycles back to the names under which they ran.

Increasing Generations. Long-lived actions could delay a guardian from updating its generation count in the generation map at the server. To eliminate this problem we can allow an action's generation to be increased. This idea is motivated by a *deadline extension* method, developed for the physical time limit scheme, that allows an action's deadline to be postponed when it seems likely that the original deadline was too early. When a guardian wants to update its generation count at the server it must first increase the generations of all topactions that have not yet completed. Any future generation counts could be used; choosing one much later in the future

may avoid further generation increases for very long actions.[5]

To increase the generation of a topaction, a guardian sends an *increase generation* message to each guardian where a subaction of the topaction may still be running; the message contains the new generation and identifies the subaction whose generation should be increased. Also, it sends an *increase generation* message to each guardian of committed descendants of the topaction. This set of committed descendants' guardians corresponds to the *plist* (for "participant list") that Argus stores for each active action. The message to members of the plist contains the new generation, the topaction's identifier, and the aborted set of the sending guardian. The recipient can use the topaction's identifier to identify the local actions whose generations must be increased and it can use the aborted set to avoid increasing the generations of orphans. The recipient must union the aborted set with its own before increasing the generations of non-orphaned actions (or releasing locks of orphans).

When a guardian receives an *increase generation* message, it updates the generation of the subaction if it is still active and sends an *increase generation* message to all guardians where subactions of the increased subaction may still be running, as well as to all members of the increased subaction's plist. In this way information about the new generation reaches all guardians running descendants of the increased topaction and holding locks on behalf of its committed descendants.

When a guardian has successfully increased the generations of a local subaction and its descendants, it returns an acknowledgement message to the sender of the *increase generation* message. (Guardians of active descendants must wait for acknowledgements from all members of the plist. Guardians with no active descendants return the acknowledgement after increasing the generations of all relevant lock holders). Once the topaction's guardian has received all acknowledgements, it may safely assume that the topaction and all its descendants now have the new generation.

There is a race condition in the above method for increasing generations. What if a guardian thinks that a remote subaction is still running because it has not yet received a commit message when, in fact, the commit message is in transit? The guardian of the remote action will no longer have the plist for the action. We can correct this as follows. While a guardian is waiting for an *increase generation* acknowledgement from the guardian of a supposedly active descendant, if it receives a commit message for that descendant, then it sends *increase generation* messages out to all members of the committing action's plist. If a guardian is in the process of performing a generation increase for an action that commits, it must still fix the generation count for the action locally and send an acknowledgement for itself but it need not wait for acknowledgements from members of the plist since the parent's guardian will handle it.

[5]Note that generation increases solve a different problem than that caused by delayed two-phase commits. Generation increases will require comunication with all guardians visited by committed or running descendants of a topaction, while the reason for the delay in a two-phase commit is exactly the lack of ability to communicate.

We must make sure that the acknowledgements for active actions are distinguishable from those for committed actions because the acknowledgement could overtake the commit message in the situation just described, and the parent must know to wait for the commit message and then send messages to the members of the plist.

4.3 Interactions With Orphan Detection

Our commit protocol for nested actions interacts with orphan detection in two ways:

1. For the protocol to guarantee the eager diffusion semantics, crash orphan detection must be performed.

2. The data maintained and propagated for the protocol subsumes the data needed to detect abort orphans in the Argus abort orphan detection algorithm [Liskov *et al.* 1987c].

We discuss these interactions in the following sections.

4.3.1 Crash Orphans

A crash orphan arises in Argus when a guardian storing volatile data (locks and versions) depended upon by some active action crashes. The volatile data is lost as a result of the crash, preventing the action from ever committing. As the example in Figure 4.3 illustrates, the existence of crash orphans can interfere with the ability of the protocol to support eager diffusion, even for non-crash orphans.

In the example, $A.2$ is a crash orphan since it could depend upon information provided to its parent by $A.1$ and lost in the crash at G_3. After $B.1$ obtains the lock on X at G_3, it could know that $A.1$ effectively never accessed X (this is true as a result of the crash). Then $B.2$ could expect the lock on Y to be immediately available at G_2; for example, it would expect this if there is a constraint on X and Y that X is always locked before Y. So the eager diffusion semantics implies that $B.2$ can get its lock. However, when G_3 recovers from its crash, it has no information that $A.1$ ever ran there. There is no information for $B.1$ to pick up at G_3 and transmit in its RPC reply message that allows any future actions that know about $B.1$'s commit to know about $A.1$'s effective abort. Thus the protocol, as defined in Chapter 3, could not guarantee the eager diffusion semantics in this situation.

If the Argus crash orphan detection algorithm is being used, then the message sent to G_2 creating $B.2$ carries enough information to detect that $A.2$ is a crash orphan. This causes $A.2$'s lock to be released before $B.2$ attempts to obtain it. Briefly, the reason that $A.2$ can be detected as a crash orphan is that when $A.1$ committed to A it carried the current *crash count* for G_3 with it. This crash count was then carried by $A.2$ to G_2 on its RPC call message. When G_3 recovered from its crash, its crash

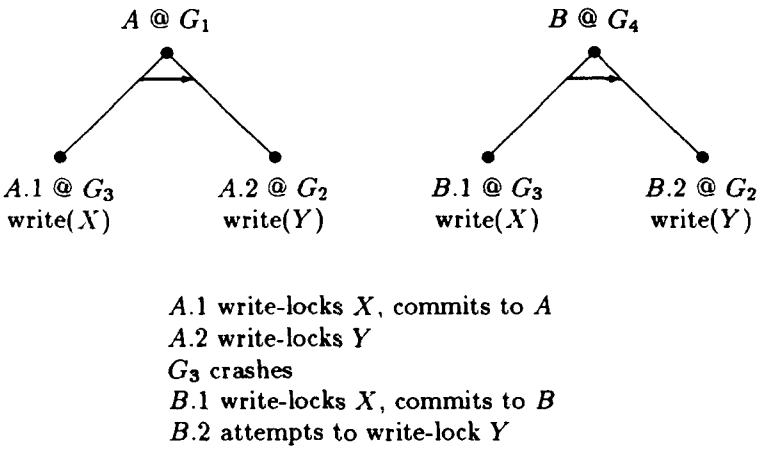


Figure 4.3: The effect of crash orphans on the ability of the protocol to ensure eager diffusion.

count was incremented. Thus when $B.1$ committed to B, the crash count that was returned to B for G_3 was higher than the one returned by $A.1$. The higher crash count for G_3 was then sent along with the call message for $B.2$ to G_2. The obsolete crash count for G_3 associated with $A.2$ identifies $A.2$ as a crash orphan.

We would like to be able to identify the precise property of the Argus crash orphan detection algorithm that is relied upon by our protocol. It is almost certainly stronger than the orphan detection correctness property requiring that orphans not see inconsistent states. Both informal and formal statements of the property would be useful, allowing us to identify other crash orphan detection algorithms that would work with our protocol and to prove rigorously that the Argus protocol really does do enough to ensure the correctness of our protocol. This question remains a topic for future research.

4.3.2 Abort Orphans

Abort orphans do not interfere with the commit protocol in the way that crash orphans do. This is because the protocol actually maintains enough information to recognize abort orphans. The information maintained in the aborted sets at guardians and in messages for the commit protocol is a superset of the information kept in the *done* data structure in the Argus orphan detection algorithm. While *done* need only contain identifiers of aborted actions that may have active remote descendants, the aborted sets must contain identifiers of aborted actions that may have committed

remote descendants as well. The abort orphan algorithm requires less information because once an action is committed, there is no chance of it seeing an inconsistent state when one of its ancestors aborts.

This observation naturally leads to the idea that in a system containing both the Argus abort orphan detection algorithm and our commit protocol, the implementations of the algorithms could be optimized to share information. This sharing must then be taken into account in considering the overhead cost of the commit protocol in terms of performance. If the abort orphan detection algorithm is already in use then the additional cost of the protocol is less than it may first appear to be.

4.4 Early Release of Read Locks

The optimization of releasing read locks early, during the first phase of two-phase commit, interacts with our protocol in that it enables actions to have additional knowledge about the execution status of other actions.[6] Suppose an action A that holds a read lock on an object X enters phase one of its two-phase commit. When the guardian of X receives a prepare message for A, it releases A's read lock. Now another action B is free to obtain a write lock on X. By obtaining the write lock, B learns that one of the following is true: either A never accessed X, or the access by A was effectively aborted, or A has started its two-phase commit (and possibly finished it). With this knowledge, B could know that other read locks held by A should be available, and it could even know that write locks held by A should be available, as we explain below. As our protocol is described up to this point, the guardians where A's locks are held will not necessarily have as much knowledge as B does if read locks are released early.

There are two changes to our protocol that will ensure support for the eager diffusion semantics when read locks are released early.

- A coordinator includes its aborted set and topactions from its committed set on phase one prepare messages.

- We introduce a new set at each guardian called *finishing*. A topaction's identifier is inserted into its guardian's finishing set when it begins two-phase commit. Finishing sets are sent on RPC call and reply messages, lock propagation messages, and phase one prepare messages; the entire finishing set for the sending guardian is propagated.

[6] In fact, the optimization and the accompanying discussion apply to other locking modes that are similar to reads. The important characteristic of the lock mode that allows locks to be released during phase one is that the effect on the object being locked is the same whether the lock holder ultimately commits or aborts.

An action identifier may be removed from the finishing set at a guardian as soon as it appears in the aborted or committed set. Garbage collection of finishing sets can be handled in the same way as for aborted and committed sets.

Three examples will help to illustrate how the modified protocol supports the eager diffusion semantics when actions can learn things by obtaining locks that were released early. The first example shows why the finishing set is needed, and the second and third examples demonstrate the reasons for sending the aborted and committed sets, respectively, in phase one messages.

The first example is illustrated in Figure 4.4. The scenario is as follows. Topaction A at guardian G_1 has two local descendants, $A.1$ and $A.2$. $A.1$ and $A.2$ each have a single remote child, $A.1.1$ at guardian G_2 and $A.2.1$ at guardian G_3, that have obtained read locks on objects X and Y, respectively. A starts its two-phase commit. As a result, G_2 receives a prepare message for A and releases $A.1.1$'s read lock on X. Topaction B starts running at G_2 and obtains a write lock on X. We assume that B knows that $A.2.1$ always locks Y only after $A.1.1$ has locked X and committed up to A. Thus, when B obtains the write lock it knows that either $A.1.1$ never accessed X, or $A.1.1$ was effectively aborted, or A started two-phase commit. $B.1$ is then created at G_3. Since $B.1$ knows everything that B knows, $B.1$ can expect to obtain a write lock on Y.

Under the original version of our protocol, if G_3 has not yet received a prepare message for A by the time $B.1$ requests the write lock, G_3 might not have enough information to release $A.2$'s lock (it would be able to release the lock if $A.2.1$ had aborted locally). With the above modifications, G_3 will always have enough information. Contained in the prepare message for A received by G_2 are the finishing, aborted, and toplevel committed sets of G_1, which G_2 unions in with its own sets. The finishing set contains A and the aborted set contains an ancestor of each of A's aborted descendants. These sets will be propagated to G_3 on the RPC create message for $B.1$. If $A.2.1$ has effectively aborted due to the abort of $A.2$, then G_3 will be able to determine this from the aborted set and release the read lock on Y. If $A.2.1$ has committed up to A, then G_3 will use the fact that A is in the finishing set (and thus has started two-phase commit) to release the read lock on Y. In the latter case, G_3 will also record that it is expecting a prepare message for A.

In this first example, the finishing set alone would actually have sufficed. If $A.2$ had aborted but G_3 did not have access to a list of A's aborted descendants, then G_3 could still release $A.2.1$'s read lock based on the fact that A has started two-phase commit. It would then be expecting a prepare message for A that might never arrive. At some later point, G_3 would either learn that $A.2$ aborted through information arriving in a later committed or aborted set, or it could query to G_1 to determine the fate of $A.2$. In our second example, the inclusion of the aborted set on a phase one prepare message will be crucial to the correctness of the protocol.

For the second example, we use the same action structure as in Figure 4.4, but this

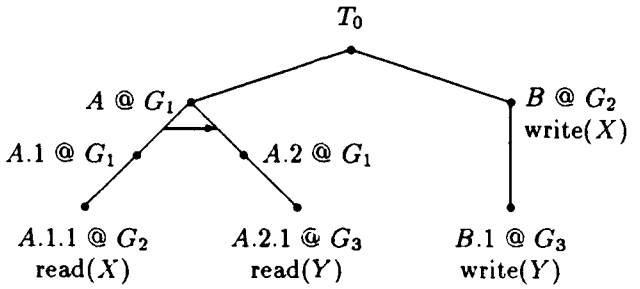


Figure 4.4: Example 1: The effect of early release of read locks.

time $A.2.1$ will obtain a write lock on Y at G_3 instead of just a read lock. Consider the following (admittedly contrived) scenario.

- Action A runs at guardian G_1. A has children $A.1$ and $A.2$. $A.1$ has child $A.1.1$ which runs at G_2 and gets a read lock on an object X. $A.2$ has child $A.2.1$ which runs at G_3 and gets a write lock on an object Y. $A.1.1$ and $A.2.1$ are such that $A.1.1$ must commit up to A before $A.2.1$ ever runs. Furthermore, A has the particularly strange behavior that it will only ever try to commit if $A.2$ aborts.

- Action B runs at guardian G_2. It has a descendant $B.1$ that runs at G_3. B obtains a write lock on X and $B.1$ obtains a write lock on Y.

- Execution proceeds as follows:

  1. A starts running at G_1.
  2. B starts running at G_2.
  3. $A.1$ starts running at G_1 and creates $A.1.1$ at G_2.
  4. $A.1.1$ obtains a read lock on X at G_2, and then commits up to $A.1$, which commits up to A.
  5. $A.2$ starts running at G_1 and creates $A.2.1$ at G_3.
  6. $A.2$ obtains a write lock on Y at G_3 and then commits to $A.2$.
  7. $A.2$ aborts.
  8. A requests to commit.
  9. A's two phase commit starts. G_2 receives the phase one message and releases $A.1.1$'s read lock. G_3 is not a participant in the two-phase commit since $A.2$ aborted.
  10. B obtains a write lock on X at G_2.

11. $B.1$ requests a write lock on Y.

The first question is whether, in step 9 of the execution, $B.1$ could expect to be able to obtain the write lock on Y. The answer is "yes." When B obtains the write lock on X in step 8 it knows that one of the following is true:

1. $A.1.1$ has not yet accessed X. In this case there will be no current lock holder for Y since $A.2.1$ will not attempt to get the lock on Y until after $A.1.1$ accesses X and commits up to A.

2. $A.1.1$ effectively aborted. If $A.2.1$ has not yet accessed Y, then the lock is available. Otherwise, $A.2.1$ must also be effectively aborted, since $A.1.1$ would have had to commit up to its least common ancestor with $A.2.1$ for $A.2.1$ to ever access Y in the first place. So if $A.1.1$ is effectively aborted, then $A.2.1$ effectively never accessed Y.

3. A started two-phase commit. A would only attempt to commit if $A.2$ was aborted.

Thus in each case, $B.1$ can reason that it should be able to get the lock.

Using our modified protocol, G_3 will be able to grant $B.1$'s lock request. G_1 includes its aborted set in the prepare message sent to G_2. This aborted set includes representatives for each of A's aborted descendants. G_2 unions the aborted set in the message with its own. Then in the create message for $B.1$, G_2 includes its aborted set. Thus G_3 has enough information to determine that $A.2.1$ was effectively aborted. Note that the finishing set received by G_3 will contain A, but this is not enough information to release $A.2.1$'s write lock on Y. A write lock may not be released until a guardian knows the true outcome of two-phase commit because the guardian must know whether to make changes to the object permanent or discard them at the time it releases the lock.

The third example, illustrated in Figure 4.5, presents a scenario where the early release of read locks requires the information about committed topactions known to an action undergoing two-phase commit to be propagated to all guardians that prepare the action. In this situation, action A is the topaction that undergoes two-phase commit. We assume that action A will only ever request to commit if it observes another action C to have committed (it could do this if C wrote X, for example). A starts two-phase commit after running two subactions, $A.1$ and $A.2$, that obtain read and write locks on objects X and Y at guardians G_2 and G_3, respectively. Guardian G_2 receives a prepare message for A and releases $A.1$'s read lock on X. After $A.1$'s read lock on X is released, action $B.1$, a subaction of topaction B, is able to obtain a write lock on X. At the point that $B.1$ obtains the write lock, it knows that one of the following is true about A:

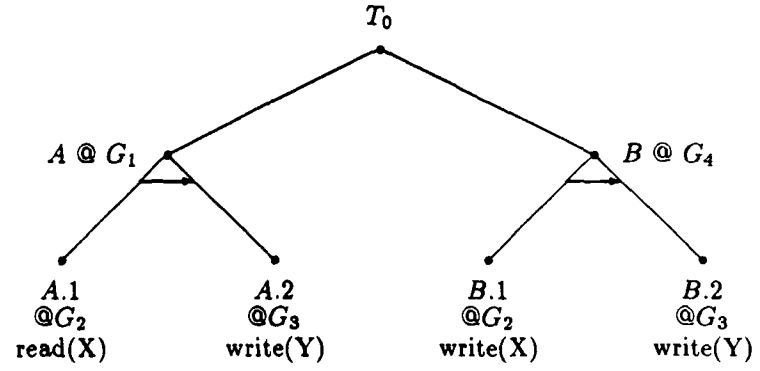


Figure 4.5: Example 3: The effect of early release of read locks.

- Either $A.1$ never obtained the lock on X, or
- $A.1$ was effectively aborted, or
- A has started two-phase commit.

B could then reason as follows:

- If $A.1$ never accessed X, or some ancestor of $A.1$ below $lca(A.1, A.2)$ was aborted, then $A.2$ never accessed Y (since it will only do so after $A.1$ commits up to their least common ancestor). Thus $B.2$ should be able to obtain the write lock on Y.
- If $A.1$ effectively never accessed X due to the abort of an ancestor of $lca(A.1, A.2)$, then $A.2$ effectively never accessed Y (because that aborted ancestor is also an ancestor of $A.2$). Thus $B.2$ should be able to obtain the write lock on Y.
- If A has started two-phase commit, then $B.2$ may not be able to access Y.

Now, when $B.2$ tests the lock on Y and finds that it cannot obtain it, B can conclude that A has started two-phase commit. This implies that C has committed. Thus, if B or one of its descendants tries to access an object locked by C, they can expect to obtain the lock. The only way to ensure that the system will be able to grant a lock to B that is still held by C (in the case that news of C's commit has not yet reached all guardians) is for B to carry along the information that C committed. This information is present in the committed set at A's guardian since A has observed C's commit.

Note that this last example relies upon B's ability to test a lock without waiting for it. As we mentioned in Chapter 2, lock testing operations are usually provided for

construction of user-defined atomic types and must be used carefully if serializability of actions is to be preserved. We think that when lock tests are used in the prescribed manner, commit information is not required to be piggybacked on phase one prepare messages. At present we do not know how to prove this.

Admittedly, the second and third examples are contrived, and we would be surprised to see these scenarios arise in practice. However, their existance implies that if we did not change the protocol in some way when read locks are released early then we could not claim that the protocol ensures eager diffusion. Also, there may be simpler, more natural examples than these, though we have not yet found any.

4.5 Improving the Performance of Lazy Diffusion

Lacking implementation data and more information about the nature of applications that will be running in our systems, we do not know at this point whether our commit protocol can be made to perform well enough to have an acceptable cost. However, even if the protocol proves impractical, ideas from the protocol may be still be useful in improving the performance of protocols that support only weaker semantic properties such as the lazy diffusion semantics. In particular, handling of commits and aborts of nested actions in Argus could benefit from ideas in our protocol.

4.5.1 Eliminating Information

The major efficiency problem with the full protocol is that of garbage collecting information about commits and aborts of topactions; the growth rate of the committed set is of particular concern. If we are willing to abandon the eager diffusion semantics, we can consider protocols similar to the one we proposed but where identifiers of committed topactions are no longer stored in committed sets or sent in messages. We may then choose among a range of possiblities for the amount of information propagated about commits and aborts. In choosing, we can trade off storage space, cost of garbage collection, and overhead incurred by larger messages against semantic support and the cost of querying.

We still have a garbage collection problem in all cases considered below. However, the rate of growth of information will be significantly smaller and thus the problem is more amenable to well-known solutions. For the fully optimized version of our protocol, under the assumptions that aborts are relatively infrequent and that large groups of committed concurrent subactions with remote descendants are rare, we can predict that the amount of information about subactions in the committed and aborted sets should not be very large. Even if aborts are common for subactions, the ability to replace subactions in the aborted sets by their ancestors should keep the set at a managable size.

The chart in Figure 4.6 illustrates a range of possible combinations of commit

		M.aborted			
		none	descendants of $top(A)$	descendants of $top(A)$ and topactions	M_A
M.committed	none	✓ *lazy diffusion*	✓	✓	✓
	descendants of A'	×	✓	✓	✓
	descendants of $top(A)$	×	✓	✓	✓
	M_C	×	×	×	✓ *eager diffusion*

Figure 4.6: Combinations of commit and abort information sent in a message M about action A. A' is an ancestor of A that is a proper descendant of $top(A)$. M_C and M_A are the comitted and aborted sets sent in the full protocol. Crosses indicate invalid combinations.

and abort information. The row and column headings indicate the different amounts of commit and abort information, respectively, that could be sent in messages. The largest amount indicated in each case is the information that we have proposed to send in the full protocol to support the eager diffusion semantics. The lesser quantities are subsets of the information sent in the full protocol. Not all combinations of commit and abort information make sense.

The alternatives that are crossed out in the chart are combinations that result in an incorrect protocol. In those cases too little abort information would be sent relative to the amount of commit information. Whenever an action is included in a committed set, all of its aborted descendants must be represented in the corresponding aborted set, either by being in the set themselves or by having an ancestor in the set. Otherwise we might infer that some descendant of the committed action is committed up to the action when, in fact, the descendant has an aborted ancestor.

All of the remaining alternatives result in correct protocols. According to the semantics that can be supported, they range from lazy diffusion, where no commit or abort information is sent and lock propagation is entirely query driven, to eager diffusion, where the information sent is as specified in our protocol. It is difficult to characterize precisely the semantics supported by the choices in between these two extremes, although we can say something about the inferences that can be made by a guardian receiving the information.

The first row of the chart eliminates the committed set entirely. Thus, a guardian receiving a message about an action A with one of the indicated aborted sets will not be able to infer the commits of any actions that are concurrent with A. For all choices of abort information, the guardian can still infer commits for descendants of $top(A)$ that run sequentially before A, and it can infer that any descendant of an action in the aborted set is effectively aborted.

In the second and third rows of the chart, we send a limited amount of commit information. When sending a message about an action A, the idea is to choose some ancestor, A', of A and just send commit information for descendants of A'. The choice of A' could vary depending upon the amount of information that we are willing to add to a message. In the third row, A' is the topaction of A. The receiving guardian will then be able to infer commits for those relatives of A that run concurrently with it and are included in the committed set, as well as for all descendants of $top(A)$ that run sequentially before A.

Another idea when sending a message about an action A is to send abort information only for descendants of an ancestor, A', of A, as we do for commit information in the second row. This turns out to be somewhat more complicated than the other choices for abort information. Once we limit the abort information to descendants of A', we can no longer use that information to make inferences about non-descendants of A'. The reason for this is related to the reason for the crossed-out choices in the chart being incorrect: in order to infer commits for descendants of some action, all

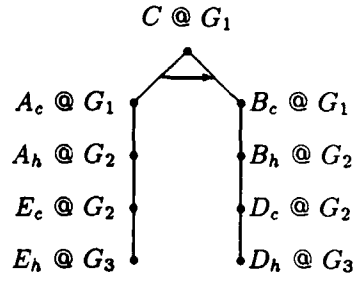


Figure 4.7: An action tree to illustrate the Argus optimization.

aborted descendants of the action must be represented in the aborted set. This is easy to deal with for a single level of propagation; in the message for action A that includes abort information for descendants of A', we simply note that A' is the cut-off point. Then the receiving guardian knows not to make incorrect inferences upon receiving the message. However, if the receiver then unions this abort information into its own aborted set, it must be careful to remember the cut-off point for the information in order to propagate correct information in future calls. This complicates the data structures and algorithm to an extent that does not seem worth the trouble.

Although multiple-level propagation is not practical when abort information is cut-off below the topaction, the single-level propagation of such abort information can be useful. The current Argus implementation uses such an optimization.[7] When a handler call is made to a guardian that has already been visited by some prior sequential descendant of a local ancestor of the call action, then some abort information may be piggybacked on the call message. This abort information can be used to propagate locks held by prior sequential relatives of the handler action. Figure 4.7 illustrates a situation in which this optimization applies. When call action B_c creates handler action B_h at guardian G_2, it will notice that its ancestor C has a descendant A_h that has already run at G_2. B_c will then piggyback information about C's aborted descendants on the call message for B_h. So if A_c had aborted, for example, then any locks held by A_h could be released and given to B_h. If A_h has instead committed up to C, G_2 will be able to infer this and propagate its locks to B_h.

The optimization is applied for a single level of propagation; the receiving guardian uses the information in the aborted set of the message to release locks and then discards the information. Thus, in our example, the abort information would not be propagated to G_3 on the call message for D_h even though it could be useful if D_h requests a lock held by E_h. In Argus, guardians do not keep aborted sets explicitly—

[7]The optimization in Argus predates our work.

instead, the abort information is obtained from the *aborts list* that is part of the state associated with a running action. The optimization for Argus was developed to solve a particular problem that was noticed during the development of an Argus application. The Argus designers have observed that the optimization has been effective in practice, and that the Argus protocol could be modified to piggyback more complete action information on calls, if it seems desirable [Liskov *et al.* 1987a]. The alternatives that we have considered here are generalizations of the Argus optimization.

4.5.2 Sequential Inference for Topactions

In Argus it is possible to write topactions that are sequential. That is, a later topaction will not be started until the topactions before it have at least completed the first phase of their two-phase commit. Once we have decided not to store complete commit and abort information for top-level actions, and thus are not supporting the eager diffusion semantics, the possibility of sequentiality for topactions becomes an additional source of useful information.

In order to do sequential inference for topactions, we must be able to determine whether two topactions are sequential based upon their action identifiers. The current implementation of Argus does not support this, but it is not difficult to imagine how it could be supported. One way to implement the action identifiers would be to introduce a new field for a topaction identifier that identified its *sequential group*. All topactions that are always run in sequence would be part of the same sequential group, and would be in a different sequential group from any other topaction. An extra counter at each guardian would provide the necessary names for sequential groups at that guardian.

The sequential inferences that can be made for topactions depend upon how much information is retained in the aborted set. First, assume that we retain no information about topactions in aborted sets. Then, given that we can determine that two topactions are sequential based on their identifiers, we can make inferences about aborts of actions. In particular, if a later sequential topaction is running, we can infer that an earlier sequential topaction must have finished the first phase of its commit protocol, and the decision to commit or abort it has been made by its home guardian. Thus, if a guardian holds a lock on behalf of a subaction of some topaction A, and is not a participant in A's two-phase commit, then a request for the lock by some later sequential topaction B indicates that the lock holder has been orphaned. If the lock holder had committed up to A, then the guardian would either be a participant awaiting the completion decision or it would have already released the lock.

As described above, no participant in A's two-phase commit can infer anything based on the fact that B is running. B could have started running whether A aborted or committed at the end of phase one and, without more information, the participant cannot determine the outcome. To make sequential inference for topactions even more

useful we could insert identifiers of aborted sequential topactions into the aborted sets at their guardians. (This must be done before any locks are released or later sequential actions are created.) We then include in the aborted sets of messages for B's descendants the topactions in B's guardian's aborted set that are sequentially before B. A participant in A's two-phase commit that is visited by a descendant of B can use the information to infer A's completion status and release locks held by A or other actions prepared for A's commit.

Note that this optimization for sequential topactions is only worth considering in the absence of the eager diffusion semantics. In the full protocol, whenever B creates a remote subaction, A will be included in either the committed or aborted set in the message that creates the subaction.

Chapter 5

Formalizing Eager Diffusion

In this chapter we present a formal definition of eager diffusion. The correctness condition set forth here will be used in the next chapter to prove formally that the unoptimized protocol presented in Chapter 3 is correct.[1] We believe that the formalization is valuable for a number of reasons. First, the protocol, even in unoptimized form, is non-trivial. It is very difficult to give correctness arguments that are both conclusive and informal. The correctness arguments in Section 3.5 were intended to give the reader some intuition about why the protocol should work, but were not intended to be complete. Second, the correctness condition—that the protocol guarantee eager diffusion—is itself non-trivial. It is easier, given a formal model, to judge whether the statement of the correctness condition actually captures our intent. Of course, in using a formalization we run the risk that the model does not correspond correctly to the actual system we are trying to describe. However, we can reason about the correspondence of the model to reality independently of the particular protocol that we are modelling. Third, it is clear that there are interactions between our protocol and other protocols in the systems we study. In particular, we have mentioned interactions with orphan detection, top-level commit protocols, and concurrency control protocols. By formalizing the different protocols it becomes easier to study and prove properties of their interactions. Also, it is easier to isolate the properties of the protocols that affect their interactions with other protocols, independently of their implementations in particular systems.

We have chosen as our formal model the Lynch-Merritt model of nested transactions (the "LM-model") [Lynch & Merritt 1986b]. The model is based on I/O automata, a simple formalization of communicating automata described in [Lynch & Tuttle 1987]. Each nested transaction and data object is modelled explicitly by an I/O automaton. Transactions and objects interact with each other through

[1] This chapter did not appear in the original thesis. The definition of eager diffusion was formalized, with the invaluable assistance of Alan Fekete, after the thesis was submitted. The proof in the next chapter has been reworked somewhat to incorporate the definition, and a speculative section that appeared in the thesis concerning a possible approach to formalizing eager diffusion has been removed.

a controller, which is another I/O automaton. The controller can be thought of as the underlying "runtime system" that, for example, supports creation and completion of transactions and handles communication among them. It is in the controller that we embed our commit protocol.

Our choice of the LM-model for our formal work was influenced by a number of factors. An important one is that the model is available and well-defined, and we have previous experience working with it. Another consideration is that there is ongoing work in using the LM-model to describe orphan detection protocols, concurrency control algorithms, and commit protocols. By choosing a common formalization for our algorithm, we hope to facilitate later study of the interactions among the different algorithms in a formal setting. One drawback to the model is that it does not include crashes of transactions and objects. The authors of [Lynch & Merritt 1986b] are currently studying how best to incorporate crashes into the formal model.

We begin the Chapter by reviewing the basics of the LM-model. Large parts of Section 5.1 are taken verbatim from the description in [Herlihy *et al.* 1987], with all changes that we have introduced into the model so noted. We then proceed to give our formal definition of eager diffusion, first introducing the subsidiary notion of *dependency relations* on events in the formal model.

5.1 The Formal Model

This section describes the formal foundation upon which we base our descriptions and proofs. There are two parts to the model: I/O automata and nested transaction systems. I/O automata are a general mechanism for describing concurrent systems. Nested transaction systems are particular kinds of concurrent systems, with particular structures and conventions, that may be modelled using I/O automata. There are many different aspects of nested transactions that can be modelled. If we were interested in studying concurrency control algorithms, for example, then we would want our model to reveal the concurrency among transactions explicitly. On the other hand, if we were interested in studying applications that were built using transactions, we would not need to model the concurrency because users of transactions need not reason about other concurrent activities—one of the purposes of atomicity is to mask concurrency. For our purposes in this chapter, we do require a model that reveals how transactions actually execute, including the concurrent interleavings of operations of different transactions. The formal systems that model these nested transaction systems with concurrency are called *generic systems*.

First we describe the basic I/O automaton model, and then we describe generic systems.

5.1.1 Basic Model

The LM-model of nested transactions is based on the I/O automaton model for concurrent systems [Lynch & Tuttle 1987, Lynch & Merritt 1986b]. The model consists of (possibly infinite-state) nondeterministic automata that have operation names associated with their state transitions. Communication among automata is described by identifying their operations. For our purposes we need only consider a special case of the general model that is concerned with finite behavior. This section reviews the relevant definitions for I/O automata.

I/O Automata. An I/O automaton $\mathcal{A}$ has components $states(\mathcal{A})$, $start(\mathcal{A})$, $out(\mathcal{A})$, $in(\mathcal{A})$, and $steps(\mathcal{A})$. Here, $states(\mathcal{A})$ is a set of states, of which a subset, $start(\mathcal{A})$, is designated as the set of start states. The next two components are disjoint sets: $out(\mathcal{A})$ is the set of *output operations*, and $in(\mathcal{A})$ is the set of *input operations*. The union of these two sets is the set of *operations* of the automaton. Finally, $steps(\mathcal{A})$ is the transition relation of $\mathcal{A}$, which is a set of triples of the form (s', π, s), where s' and s are states, and π is an operation. Such a triple means that in state s', the automaton can atomically do operation π and change to state s. An element of the transition relation is called a *step* of $\mathcal{A}$. If (s', π, s) is a step of $\mathcal{A}$, we say that π is *enabled* in s'.

The output operations are intended to model the actions that are triggered by the automaton itself, while the input operations model the actions that are triggered by the environment of the automaton. We require the following condition, which says that an I/O automaton must be prepared to receive any input operation at any time.

> *Input Condition*: For each input operation π and each state s', there exists a state s and a step (s', π, s).

An *execution* of $\mathcal{A}$ is a finite alternating sequence $s_0, \pi_1, s_1, \pi_2, \ldots$ of states and operations of $\mathcal{A}$, ending with a state. Furthermore, s_0 is in $start(\mathcal{A})$, and each triple (s', π, s) that occurs as a consecutive subsequence is a step of $\mathcal{A}$. From any execution, we can extract the *schedule*, which is the subsequence of the execution consisting of operations only. Because transitions to different states may have the same operation, different executions may have the same schedule.

If S is any set of schedules (or property of schedules), then $\mathcal{A}$ is said to *preserve* S provided that the following holds. If $\alpha = \alpha'\pi$ is any schedule of $\mathcal{A}$, where π is an output operation, and α' is in S, then α is in S. That is, the automaton is not the first to violate the property described by S.

Composition of Automata. We describe systems as consisting of interacting components, each of which is an I/O automaton. It is convenient and natural to view

systems as I/O automata, also. Thus, we define a composition operation for I/O automata, to yield a new I/O automaton.

A set of I/O automata may be composed to create a *system* $\mathcal{S}$, if the sets of output operations for the automata are disjoint. (Thus, every output operation in $\mathcal{S}$ will be triggered by exactly one component.) The system $\mathcal{S}$ is itself an I/O automaton. A state of the composed auotmaton is a tuple of states, one for each component, and the start states are tuples consisting of start states of the components. The set of *operations* of $\mathcal{S}$, $ops(\mathcal{S})$, is exactly the union of the sets of operations of the component automata. The set of *output operations* of $\mathcal{S}$, $out(\mathcal{S})$, is likewise the union of the sets of output operations of the component automata. Finally, the set of *input operations* of $\mathcal{S}$, $in(\mathcal{S})$, is $ops(\mathcal{S}) - out(\mathcal{S})$, the set of operations of $\mathcal{S}$ that are not output operations of $\mathcal{S}$. The output operations of a system are intended to be exactly those that are triggered by components of the system, while the input operations of a system are those that are triggered by the system's environment.

The triple (s', π, s) is in the transition relation of $\mathcal{S}$ if and only if for each component automaton $\mathcal{A}$, one of the following two conditions holds. Either π is an operation of $\mathcal{A}$, and the projection of the step onto $\mathcal{A}$ is a step of $\mathcal{A}$, or else π is not an operation of $\mathcal{A}$, and the states corresponding to $\mathcal{A}$ in the two tuples s' and s are identical. Thus, each operation of the composed automaton is an operation of a subset of the component automata. During an operation π of $\mathcal{S}$, each of the components that has operation π carries out the operation, while the remainder stay in the same state. Again, the operation π is an output operation of the composition if it is the output operation of a component—otherwise, π is an input operation of the composition.

An *execution* of a system is defined to be an execution of the automaton composed of the individual automata of the system. If α is a sequence of operations of a system $\mathcal{S}$ with component $\mathcal{A}$, then we denote by $\alpha \mid \mathcal{A}$ the subsequence of α containing all the operations of $\mathcal{A}$. Clearly, if α is a schedule of $\mathcal{S}$, $\alpha \mid \mathcal{A}$ is a schedule of $\mathcal{A}$.

The following lemma from [Lynch & Merritt 1986b] expresses formally the idea that an operation is under the control of the component of which it is an output.

Lemma 1. Let α' be a schedule of system $\mathcal{S}$, and let $\alpha = \alpha'\pi$, where π is an output operation of component $\mathcal{A}$. If $\alpha \mid \mathcal{A}$ is a schedule of $\mathcal{A}$ then α is a schedule of $\mathcal{S}$.

5.1.2 Generic Systems

In this section, we define *generic systems*, which consist of transactions, generic objects, and a generic controller. These systems model the way in which transactions actually execute in a distributed system. Transactions and generic objects describe user programs and data, respectively. The generic controller controls communication between the components, and thereby defines the allowable orders in which the transactions may take steps. All three types of system components are modelled as I/O automata.

We begin by defining a structure that describes the nesting of transactions. Namely, a *system type* is a four-tuple $(\mathcal{T}, parent, \mathcal{O}, V)$, where $\mathcal{T}$, the set of transaction names, is organized into a tree by the mapping $parent : \mathcal{T} \longrightarrow \mathcal{T}$, with T_0 as the root. In referring to this tree, we use traditional terminology, such as child, leaf, least common ancestor (lca), ancestor and descendant. (A transaction is its own ancestor and descendant.) The leaves of this tree are called *accesses*. The set $\mathcal{O}$ denotes the set of objects; formally, $\mathcal{O}$ is a partition of the set of accesses, where each element of the partition contains the accesses to a particular object. The set V is a set of *values*, to be used as return values of transactions. The tree structure can be thought of as a predefined naming scheme for all possible transactions that might ever be invoked. In any particular execution, however, only some of these transactions will actually take steps. We imagine that the tree structure is known in advance by all components of a system. The tree will, in general, be an infinite structure.

The classical transactions of concurrency control theory (without nesting) appear in our model as the children of a "mythical" transaction, T_0, the root of the transaction tree. It is convenient to introduce the root transaction to model the environment in which the rest of the transaction system runs. Transaction T_0 has operations that describe the invocation and return of the classical transactions. It is natural to reason about T_0 in much the same way as about all of the other transactions.

The internal nodes of the tree model transactions whose function is to create and manage subtransactions, but not to access data directly. The only transactions that actually access data are the leaves of the transaction tree, and thus they are distinguished as "accesses". The partition $\mathcal{O}$ simply identifies those transactions that access the same object.

A generic system of a given system type is the composition of a set of I/O automata. This set contains a transaction automaton for each *internal* (*i.e.*, non-leaf, non-access) node of the transaction tree, a generic object automaton for each element of $\mathcal{O}$, and a generic controller. These automata are described below. (If X is a generic object associated with an element $\mathcal{X}$ of the partition $\mathcal{O}$, and T is an access in $\mathcal{X}$, we write $T \in accesses(X)$ and say that "T is an access to X".)

For the rest of this chapter, we fix a particular system type $(\mathcal{T}, parent, \mathcal{O}, V)$.

Transactions. Transactions are modelled as I/O automata. In modelling transactions, we try not to constrain them unnecessarily; thus, we do not require that they be expressible as programs in any particular high-level programming language. Modelling the transactions as I/O automata allows us to state exactly the properties that are needed without introducing unnecessary restrictions or complicated semantics.

A non-access transaction T is modelled as an I/O automaton, with the following operations:

Input operations:

CREATE(T)
COMMIT(T', v), for $T' \in$ *children*(T) and $v \in V$
ABORT(T'), for $T' \in$ *children*(T)

Output operations:

REQUEST-CREATE(T'), for $T' \in$ *children*(T)
REQUEST-COMMIT(T, v), for $v \in V$

The CREATE input operation "wakes up" the transaction. The REQUEST-CREATE output operation is a request by T to create a particular child transaction.[2] The COMMIT input operation reports to T the successful completion of one of its children, and returns a value recording the results of that child's execution. The ABORT input operation reports to T the unsuccessful completion of one of its children, without returning any other information. We call COMMIT(T', v), for any v, and ABORT(T') *return operations* for transaction T'. The REQUEST-COMMIT operation is an announcement by T that it has finished its work, and includes a value recording the results of that work.

It is convenient to use two separate operations, REQUEST-CREATE and CREATE, to describe what takes place when a subtransaction is activated. The REQUEST-CREATE is an operation of the transaction's parent, while the actual CREATE takes place at the subtransaction itself. Similar remarks hold for the REQUEST-COMMIT and COMMIT operations.

We leave the execution of particular transaction automata largely unspecified; the choice of which children to create, and what value to return, will depend on the particular implementation. However, it is convenient to assume that schedules of transaction automata obey certain syntactic constraints. Thus transaction automata are required to preserve well-formedness, as defined below.

We recursively define *well-formedness* for sequences of operations of a transaction T. Namely, the empty schedule is well-formed. Also, if $\alpha = \alpha'\pi$ is a sequence of operations of T, where π is a single operation, then α is well-formed provided that α' is well-formed, and the following hold:

- If π is CREATE(T), then
  1. there is no CREATE(T) in α'.
- If π is COMMIT(T', v) or ABORT(T') for a child T' of T, then

[2] Note that there is no provision for T to pass information to its child in this request. In a programming language, T might be permitted to pass parameter values to a subtransaction. Although this may be a convenient descriptive aid, it is not necessary to include it in the underlying formal model. Instead, we consider transactions that have different input parameters to be different transactions.

1. REQUEST-CREATE(T') appears in α', and
2. there is no return operation for T' in α'.

- If π is REQUEST-CREATE(T') for a child T' of T, then
  1. there is no REQUEST-CREATE(T') in α', and
  2. there is no REQUEST-COMMIT(T) in α', and
  3. CREATE(T) appears in α'.
- If π is a REQUEST-COMMIT for T, then
  1. there is no REQUEST-COMMIT for T in α', and
  2. CREATE(T) appears in α', and
  3. for each REQUEST-CREATE(T') that appears in α', for T' a child of T, a return operation for T' appears in α'.[3]

These restrictions are very basic; they simply say that a transaction does not get created more than once, does not receive repeated notification of the fates of its children, does not receive conflicting information about the fates of its children, and does not receive information about the fate of any child whose creation it has not requested; also, a transaction does not perform any output operations before it has been created or after it has requested to commit, does not request the creation of the same child more than once, and does not request to commit before it has learned the fate of all children whose creation it requested. Except for these conditions, there are no restrictions on allowable transaction behavior. For example, a transaction can request creation of new subtransactions at any time, without regard to its state of knowledge about subtransactions whose creation it has previously requested. Particular programming languages may choose to impose additional restrictions on transaction behavior. (An example is Argus, which suspends activity in transactions until subtransactions complete.) However, our results do not require such restrictions.

Generic Objects. Generic objects are similar to the abstract objects of Argus and other "object-oriented" systems. A generic object provides a set of "operations" (not to be confused with the operations of an I/O automaton) through which transactions can observe and change the object's state. For uniformity and ease of exposition, we model each possible instance of an "operation" as a subtransaction, here called an *access transaction*. Accesses can be invoked by concurrent transactions,

[3]This restriction on transactions was not present in [Herlihy *et al.* 1987]. Instead, a precondition on the COMMIT operation of the generic controller produced essentially the same effect. We found it more convenient for our proofs to place the restriction on transactions, as have the authors of [Herlihy *et al.* 1987] in some of their more recent work.

and transactions can abort; thus, generic objects must provide synchronization and recovery sufficient to ensure serializability of the transactions using them. For example, the particular objects studied in [Lynch & Merritt 1986b], which use an exclusive locking variation of Moss's algorithm [Moss 1981] for synchronization combined with version stacks for recovery, have been shown to be correct for non-orphan transactions [Lynch & Merritt 1986b]. Correctness of other generic objects has also been shown: read/write locking is studied in [Fekete *et al.* 1987], more general locking objects are studied in [Fekete *et al.* 1988], and objects using timestamps are discussed in [Aspnes 1987].

A generic object X is modelled as an I/O automaton, with the following operations:

Input Operations:

CREATE(T), T an access to X
INFORM-COMMIT-AT(X)OF(T)
INFORM-ABORT-AT(X)OF(T)

Output Operations:

REQUEST-COMMIT(T, v), T an access to X

The CREATE input operation starts an access transaction at the object. (Thus, it corresponds to the invocation of an instance of one of the object's "operations".) Similarly, the REQUEST-COMMIT output indicates that an access transaction has finished its work, and includes a value recording the results. The INFORM-COMMIT and INFORM-ABORT input operations tell X that some transaction (not necessarily an access to X) has committed or aborted, respectively.

As for transaction automata, we leave the executions of particular generic objects largely unspecified. However, we do assume, as for transactions, that schedules of generic objects obey certain syntactic constraints. Thus, generic objects are required to preserve well-formedness, defined recursively as follows: First, the empty schedule is well-formed. Second, if $\alpha = \alpha'\pi$ is a sequence of operations of X, then α is well-formed provided that α' is well-formed and the following hold:

- If π is CREATE(T), then
  1. there is no CREATE(T) in α'.
- If π is a REQUEST-COMMIT for T, then
  1. there is no REQUEST-COMMIT for T in α', and
  2. CREATE(T) occurs in α'.

- If π is INFORM-COMMIT-AT(X)OF(T), then
  1. there is no INFORM-ABORT-AT(X)OF(T) in α', and
  2. if T is an access to X, then a REQUEST-COMMIT for T occurs in α'.
- If π is INFORM-ABORT-AT(X)OF(T), then
  1. there is no INFORM-COMMIT-AT(X)OF(T) in α'.

These restrictions are quite basic. They state that a given access is created at *most once*, and *requests to commit* at most once, and then only if it has been created. In addition, an object should not be given conflicting information about the fate of a transaction, *i.e.*, it should not be told both that a transaction committed and that it aborted. Finally, an object X should be told that an access to X has committed only if the access actually requested to commit.

Generic Controller. The third kind of component in a generic system is the generic controller. The generic controller is also modelled as an automaton. The transactions and generic objects have been specified to be any I/O automata whose operations and behavior satisfy simple syntactic restrictions. A generic controller, however, is a fully specified automaton, particular to each system type. (Recall that we have assumed that the system type is fixed; we describe the generic controller for the fixed system type.)

Note that the generic controller defined here differs slightly from the one in [Herlihy *et al.* 1987] because of the restriction we impose on transactions that REQUEST-COMMIT not occur while returns of children are outstanding. This allows us to omit a precondition on the COMMIT operation of our generic controller that is present in the generic controller in [Herlihy *et al.* 1987].

The generic controller has seven operations:

Input operations:

REQUEST-CREATE(T)
REQUEST-COMMIT(T, v)

Output operations:

CREATE(T)
COMMIT(T, v)
ABORT(T)
INFORM-COMMIT-AT(X)OF(T)
INFORM-ABORT-AT(X)OF(T)

The REQUEST-CREATE and REQUEST-COMMIT inputs are intended to be identified with the corresponding outputs of transaction and object auotmata, and correspondingly for the output operations.

Each state s of the generic controller consists of five sets: create-requested(s), created(s), commit-requested(s), committed(s), and aborted(s). The set commit-requested(s) is a set of $\langle$transaction, value$\rangle$ pairs, and the others are sets of transactions. The initial state of the generic controller is denoted by s_0. All of the components of s_0 are empty except for create-requested, which is $\{T_0\}$. For a state s, we define returned(s) = committed(s) $\cup$ aborted(s).

The transition relation for the generic controller consists of exactly those triples (s', π, s) satisfying the preconditions and postconditions below, where π is the indicated operation. For brevity, we include in the postconditions only those components of the state s that may change with the operation. If a component of s is not mentioned in the postcondition then the component is taken to be the same in s as in s'.

- π = REQUEST-CREATE(T)

  Post: create-requested(s) = create-requested(s') $\cup \{T\}$

- π = REQUEST-COMMIT(T, v)

  Post: commit-requested(s) = commit-requested(s') $\cup \{(T, v)\}$

- π = CREATE(T)

  Pre: $T \in$ create-requested(s') $-$ created(s')
  Post: created(s) = created(s') $\cup \{T\}$

- π = COMMIT(T, v)

  Pre: $(T, v) \in$ commit-requested(s')
  $T \notin$ returned(s')
  Post: committed(s) = committed(s') $\cup \{T\}$

- π = ABORT(T)

  Pre: $T \in$ create-requested(s') $-$ returned(s')
  Post: aborted(s) = aborted(s') $\cup \{T\}$

- π = INFORM-COMMIT-AT(X)OF(T)

Pre: $T \in \text{committed}(s')$

- $\pi = \text{INFORM-ABORT-AT}(X)\text{OF}(T)$

  Pre: $T \in \text{aborted}(s')$

The controller assumes that its input operations, REQUEST-CREATE and REQUEST-COMMIT, can occur at any time, and simply records them in the appropriate components of the state. Once the creation of a transaction has been requested, the controller can create it by producing a CREATE operation. The precondition of CREATE indicates that a given transaction will be created at most once; the postcondition of CREATE records the fact that the creation has occurred. Similarly, the postconditions for COMMIT and ABORT record that the operation has occurred. INFORM-COMMIT and INFORM-ABORT operations can be generated at any time after the corresponding COMMIT and ABORT operations have occurred.

The precondition for the COMMIT operation ensures that a transaction only commits if it has requested to do so, and has not already returned (committed or aborted). Note that our well-formedness conditions on transactions guarantee that all children whose CREATEs have been requested by the committing transaction have returned.

The precondition for the ABORT operation ensures that a transaction will be aborted only if a REQUEST-CREATE has occurred for it and it has not already returned. There are no other constraints on when a transaction can be aborted, however. For example, a transaction can be aborted while some of its descendants are still running.

The following lemma states some simple invariants relating schedules of the generic controller to the states that result from applying them to the initial state.

Lemma 2. Let α be a schedule of the generic controller, and let s be a state that can result from applying α to the initial state s_0. Then the following conditions are true.

1. T is in create-requested(s) exactly if α contains a REQUEST-CREATE(T) operation.
2. T is in created(s) exactly if α contains a CREATE(T) operation.
3. (T, v) is in commit-requested(s) exactly if α contains a REQUEST-COMMIT(T, v) operation.
4. T is in committed(s) exactly if α contains a COMMIT operation for T.
5. T is in aborted(s) exactly if α contains an ABORT(T) operation.
6. $\text{aborted}(s) \cap \text{committed}(s) = \emptyset$.

Proof: Straightforward. ■

Generic Systems. The composition of transactions with generic objects and the generic controller is called a *generic system* (of the given system type). The non-access transactions and the generic objects are called the system *primitives*. The schedules of a generic system are called *generic schedules*.

Define the *generic operations* to be those operations that occur in the generic system: REQUEST-CREATEs, REQUEST-COMMITs, CREATEs, COMMITs, ABORTs, INFORM-COMMITs and INFORM-ABORTs. For any generic operation π, we define $location(\pi)$ to be the primitive at which π occurs. (Each operation occurs both at a primitive and at the generic controller; no operation, however, occurs at more than one primitive.) For a generic operation π that occurs at a transaction, we define $transaction(\pi)$ to be that transaction.

A sequence of generic operations is called *well-formed* provided that its projection on each generic primitive (transaction and generic object) is well-formed.

Lemma 3. Every generic schedule is well-formed.

Proof: As in [Herlihy *et al.* 1987]. ■

Serial Correctness. In describing generic objects we mentioned that they must provide synchronization and recovery sufficient to ensure serializability of the transactions using them. Ultimately, we would like to prove that the generic systems we use are correct in some sense that describes how transaction systems are supposed to behave. The usual notion of correctness in much of the database literature on transactions is *serializability*. In order to handle nested transactions and aborts, the notion of serializability is generalized in the definition of correctness for generic systems presented in [Lynch & Merritt 1986b].

While serial correctness of generic schedules is important, we are not concerned with it in this thesis. The correctness condition for our protocol, which will be presented in Section 6.1, has nothing to say about serializability. Since the systems we define for our proof are generic systems, the serial correctness results for generic systems will apply to our systems as well.

For a presentation of serial correctness, and a proof that particular generic systems are serially correct, we refer the reader to [Lynch & Merritt 1986b].

5.2 Relating the Model to Reality

There are a number of differences between the formal model that we have just described and the informal model of computation that we presented in Chapter 2 and used in Chapters 3 and 4. Two general differences between the models are:

- *Guardians.* In our informal model, guardians are entities that contain transactions, objects, and the part of the runtime system that manages those transactions and objects. To incorporate guardians into the formal model we could

partition the set of transactions and object primitives so that two primitives are in the same partition exactly when they reside at the same guardian. We would also have to "distribute" the controller by separating it into parts, one to handle each partition of transactions and objects. The controller for each guardian would no longer have access to the states of other controllers for other guardians. We have not taken this step in the model for our protocol. However, it should be fairly straightforward to do so since, for the most part, our descriptions rely only on information that would be locally available. This is discussed further in the last section of this Chapter.

- *Access Transactions.* Our informal model does not distinguish a class of transactions that may create children but not access objects from a class of transactions that may access objects but not create children. All transactions in our informal model may do both. However, it is not difficult to imagine how the informal model of transactions could be mapped to the formal model. We can view the invocation of an operation on an object in the informal model as a CREATE for an access transaction; the response to the invocation is like a COMMIT for the access transaction. In reality, rather than being performed in a new nested transaction, these operations are simply performed as part of the parent transaction.

There are other differences between the formal and informal models relating to the ways in which we describe our protocol in each model. In the informal model, committed and aborted sets are associated with guardians. In the formal model we associate committed and aborted sets with operations (and hence, with transactions or objects, according to the locations of the operations). We would consider a "guardian's" committed (or aborted) set—if there were guardians in the formal model—to be the union of the committed (or aborted) sets for all transactions and objects residing at that guardian. Thus the formalization of the protocol maintains information at a finer granularity than does the actual protocol. Associating the sets with operations rather than directly with transactions or objects gives us even finer granularity. By maintaining information at a fine granularity we are able to specify the minimal information propagation necessary in the protocol to ensure the correctness condition. In the actual protocol we use a coarser granularity to eliminate redundantly stored information, reducing space costs and associated computation costs. Our formal proof still demonstrates the correctness of the coarser-grained protocol because the formalization of the protocol is nondeterministic. The description allows more information to be included in committed and aborted sets (under certain restrictions)--it simply does not require the extra information to be there.

Another difference between the formalization of the protocol and the actual protocol is that we do not directly model the sending and receiving of messages. We can view the REQUEST-CREATE, REQUEST-COMMIT, and INFORM operations as initiating

the sending of messages when the transaction or object involved in the operation is at a "remote site". Any information that must be transferred between sites for the operation is piggybacked on the messages.

One last point is that, in the formalization, we do not make assumptions about how objects use the information in INFORM operations. An INFORM-COMMIT operation for a transaction T in the formal model corresponds to information conveyed in an explicit lock propagation message saying that T committed (one such message can produce many INFORMs), or to the presence of T in the committed set of an incoming message, or to an inference made about T's commit in the informal model. Similarly, an INFORM-ABORT operation for T corresponds to an explicit lock propagation message saying that T aborted, or to the presence of T in the aborted set of an incoming message, or to an inference made about T's abort. In stating our formal correctness conditions for the protocol we simply ensure that all necessary INFORM operations have occurred by the time an access to an object occurs. We do not make any statements about the propagation or release of locks, since our systems are defined to work with all generic objects and only specific kinds of objects use locks. We assume that the objects in our systems make appropriate use of the information they receive. Clearly, we could narrow the formal specification of our system to use only the intended kinds of objects and then prove that the objects propagate or release locks as required.

5.3 Information Flow

We will make use of the notion of *dependency relations*, introduced in [Herlihy *et al.* 1987], in the correctness conditions that we define in the next section. We define two different dependency relations, similar to the *affects* relation defined in [Herlihy *et al.* 1987], to model the information flow among transactions. The first relation that we present models the flow of information about commits, while the second relation models the flow of information about aborts. The reason for using two different relations has to do with the nature of the information and our desire for efficiency in our protocol. Recall from Section 3.1 that we may characterize the commits known to a transaction A as a subset of the transactions that are visible to A. We made use of this fact in deciding what information to propagate in our protocol. Also recall that we could not characterize the aborts known to A in the same way and thus could not use the same optimizations for the amount of abort information sent in the protocol.

The dependency relations are defined in terms of events (particular operations in a schedule) rather than transactions. The definitions of the relations state the conditions under which one event affects another. Intuitively, when we say that an event of transaction A affects an event of another transaction B we mean that once the affected event of B occurs, the affecting event of A has occurred and is evidence

for some piece of B's knowledge about A. For example, if when B is created, it knows that A has committed, then the COMMIT of A affects the CREATE of B. If when B is created, it knows that A effectively never accessed an object, and if in fact A did abort, then the ABORT of A affects the CREATE of B.

5.3.1 Visible-Affects

We model the flow of information about commits with the *directly-visible-affects* dependency relation.

Definition 4. Let α be a sequence of generic operations, T and T' transactions, and X a generic object. We say that T' *is locally visible to* T *at* X *in* α if, for each ancestor T'' of T' that is a proper descendant of $lca(T, T')$, an INFORM-COMMIT-AT(X)OF(T'') occurs in α.

Define the relation *directly-visible-affects*(α), for a sequence α of generic operations, to be the relation containing the pairs (ϕ, π) of events[4] such that ϕ occurs before π in α, and at least one of the following holds:

- $location(\phi) = location(\pi)$ = an object X, π is an output operation (i.e., REQUEST-COMMIT), $T = transaction(\pi)$, $T' = transaction(\phi)$, and T' is locally visible to T at X in the prefix of α ending with π.
- $location(\phi) = location(\pi)$ = a transaction and π is an output operation.
- ϕ = REQUEST-CREATE(T) and π = CREATE(T)
- ϕ = REQUEST-COMMIT(T, v) and π = COMMIT(T, v)
- ϕ = REQUEST-CREATE(T) and π = ABORT(T)
- ϕ = COMMIT(T, v) and π = INFORM-COMMIT-AT(X)OF(T)
- ϕ = ABORT(T) and π = INFORM-ABORT-AT(X)OF(T)

Define the relation *visible-affects*(α), for a sequence α of generic operations, to be the transitive closure of *directly-visible-affects*(α). If the pair (ϕ, π) is in the relation *directly-visible-affects*(α), we say that ϕ *directly-visible-affects* π in α, and similarly for *visible-affects*(α).

The idea is that ϕ directly-visible-affects π if they both occur at the same transaction (and π is an output, since inputs can always occur), or if they both occur at the

[4] Formally, an *event* is a pair (i, π), where i is a positive integer and π is an operation. An event (i, π) is said to *occur* in α if the i^{th} element of α is π, and the event (i, π) is an *instance* of π. To avoid becoming entangled in notation, in this presentation we will not be overly formal in distinguishing operations from events. For example, we will write that an event is CREATE(T), meaning formally that its second component is CREATE(T).

same object, π is an output, and the transaction mentioned in ϕ is locally visible to the transaction mentioned in π, or if they involve different primitives but the preconditions for the controller require ϕ to occur before π can occur. This visible-affects relation is "safe" in that it may be larger than necessary for particular transactions and objects. Since we are not analyzing program texts to determine what particular transactions actually could know based on the structure of their code, we choose a relation that contains the most information that any transaction could ever know based on the rules for constructing transactions. If the operations involve different primitives, the preconditions for π do require ϕ to occur if ϕ directly affects π. If the operations occur at the same primitive, however, it might be that ϕ happens to occur before π, yet that the particular primitive does not require ϕ to occur before π.

We do assume that objects perform concurrency control in accordance with the visibility rules. A transaction executing an operation at an object should not gain commit information about other transactions that have executed operations at the object but are not yet locally visible to it. This is captured by the use of *local visiblity* in the first clause of the relation.

5.3.2 Prefix-Affects

The *directly-prefix-affects* relation describes the flow of abort information around the system. It is defined similarly to directly-visible-affects except that the first two clauses are replaced by:

- $location(\phi) = location(\pi)$ and π is an output operation.

As for the previous relation, define *prefix-affects*(α) to be the transitive closure of *directly-prefix-affects*(α). Thus ϕ directly-prefix-affects π if they both occur at the same primitive and π is an output, or if they involve different primitives but the preconditions for the controller require ϕ to occur before π can occur. This relation is even more conservative than the previous one because it assumes that an operation π can know about all previous operations at an object—not just the ones that are locally visible to it. We require a stronger assumption for aborts than for commits because there is no property similar to visibility to limit the scope of the aborts that a transaction can learn about. The prefix-affects relation is probably not the weakest possible relation that describes what a transaction can know about aborts. However, it is not clear what the weakest relation is, so we choose one that we know to be safe.

5.4 Correctness Conditions

The correctness result that we would like to prove is that our simplified protocol, as modelled by the Inference Optimized Systems to be defined in Section 6.3, guarantees

eager diffusion. In order to state and prove such a theorem we must first formalize the definition of eager diffusion.

In previous chapters, we used an intuitive definition of eager diffusion that requires a lock to be available when a transaction requesting the lock knows that it ought to be available. We could translate this intuition directly into a formal definition. Instead we choose to give a formal definition of eager diffusion that is not specific to locking systems, and then explain how the definition could be applied to locking systems.

In Section 5.3.1 (Definition 4) we gave a definition of *local visibility*, which we then used to define the *visible-affect* dependency relation. We can define a similar notion of global visiblity.

Definition 5. Let α be a generic schedule, and T and T' transactions. We say that T' is *visible* to T in α if, for each ancestor T'' of T' that is a proper descendant of $lca(T, T')$, a COMMIT(T'') occurs in α.

Next, we use the notions of visibility to define when a transaction's fate is *globally determined* with respect to another transaction and when a transaction's fate is *locally determined* at an object with respect to another transaction.

Definition 6. Let α be a generic schedule, and T and T' be accesses to X. We say that T''s fate is *globally determined w.r.t. T in α*, written $\alpha \vdash \mathcal{G}(T', T)$, if whenever CREATE($T'$) is in α, then either T' is visible to T in α, or there is some ancestor T'' of T' for which ABORT(T'') occurs in α.

Definition 7. Let α be a generic schedule, and T and T' be accesses to X. We say that T''s fate is *locally determined w.r.t. T at X in α*, written $\alpha \vdash \mathcal{L}(T', T, X)$, if whenever CREATE($T'$) is in α, then either T' is locally visible at X to T in α, or there is some ancestor T'' of T' for which INFORM-ABORT-AT(X)OF(T'') occurs in α.

To understand how these definitions relate to locks, consider two transactions, A and B, where A holds a lock on an object X that conflicts with a lock requested by B on X. Once A's fate is globally determined with respect to B, the events that will determine what happens to A's lock have already occurred. When A's fate becomes locally determined with respect to B at X, then X will have the information that allows it propagate or release A's lock and then grant B's request.

Now we define what it means for a transaction to *know* that the fate of one transaction is globally determined with respect to another transaction. The definition is based on the standard definition of knowledge[Halpern & Moses 1987], except that we just consider knowledge at the end of a run (execution) as opposed to arbitrary points in the run. The standard definition of knowledge simply says that a transaction *knows* a fact if, in every execution of the system that looks the same to the transaction, the fact is true.

Definition 8. Let α be a generic schedule, U any transaction, and T and T' accesses to X. *U knows that the fate of T' is globally determined w.r.t. T after α*, written $\alpha \vdash K_U(\mathcal{G}(T', T))$, if, for every generic schedule β such that $\beta \mid U = \alpha \mid U$, $\beta \vdash \mathcal{G}(T', T)$.

We define what it means for one system to *implement* another system.

Definition 9. A system S is an *implementation* of a system S' if all schedules of S are schedules of S'.

Finally, we can formalize the definition of eager diffusion.

Definition 10. Let S be an implementation of a generic system. S *guarantees eager diffusion* if, for all schedules α of S ending in CREATE(T) for T an access transaction to some object X where T is not an orphan in α, $\forall T' \in accesses(X).\alpha \vdash K_T(\mathcal{G}(T',T)) \Longrightarrow \alpha \vdash \mathcal{L}(T',T,X)$.

That is, for every T' such that T knows that T''s fate is globally determined with respect to T, T''s fate is locally determined with respect to T at X.

This definition is actually stronger than what is required for locking systems. There may be a T' such that T knows that T''s fate is globally determined with respect to T, but for which there is no need for T''s fate to be locally determined with respect to T at X because T' does not interfere with T's lock request. For example, T' and T may both read X; in this case it would not be necessary for X to know the outcome of T', which holds a read lock on X, in order to grant a read lock to T. Since the definition of eager diffusion for locking systems is weaker than the general definition given above, any protocol that guarantees eager diffusion under the stronger definition, guarantees it under the weaker definition as well.

Chapter 6

Correctness Proof

Using the formal correctness condition developed in the previous Chapter, we now present a formal description and rigorous correctness proof for the unoptimized protocol of Chapter 3. The proof proceeds as follows. We define three different systems of I/O automata. Systems are composed of transactions, objects, and a controller. We use the same transactions and objects in all of the systems and vary the controller. The first system most directly ensures the correctness condition, relying upon global state information to describe a constraint on accesses to objects. The second system uses local information to guarantee the correctness condition. It corresponds to the simple protocol without inference mechanisms that we used as an intermediate step in our correctness arguments in Chapter 3. Finally, the third system corresponds to the unoptimized protocol of Chapter 3, complete with sequential and descendant inferences. At each step, we prove that the later system simulates the immediately preceding system. Thus, the second system simulates the first one, and the third system simulates the second one. The third system can then easily be shown to simulate the first one by transitivity, and from this it is easy to show that the third system is also correct by our definition.

In Appendix B we define a fourth system that includes one of the optimizations described in Chapter 4, namely, the replacement of transactions in the aborted set by their aborted ancestors. We prove that this system weakly simulates the third system in that every non-access transaction sees the same thing in the fourth system as it does in the third. The weak-simulation result does not directly imply that the fourth system guarantees eager diffusion. In the appendix we discuss what is required to show that the fourth system is actually correct. We have not formally modelled any of the other optimizations suggested in Chapter 4.

6.1 Global Knowledge Systems

The most direct way to ensure that events affecting an access to an object are known at the object when the access occurs is to add preconditions to CREATEs of accesses in the generic controller that require this condition to be satisfied. We call the generic

controller modified in this way the Global Knowledge Controller (GKC). The composition of the GKC with transactions and generic objects is called a Global Knowledge System.

6.1.1 The Global Knowledge Controller

The Global Knowledge Controller (GKC) is like the generic controller, except that it has preconditions on CREATEs of accesses that delay the operation until the appropriate INFORM-COMMIT and INFORM-ABORT operations have occurred.

The GKC has the same seven operations as the generic controller. Each state s of the GKC consists of six components. The first five are the same as for the generic controller (*i.e.*, create-requested(s), created(s), commit-requested(s), committed(s), and aborted(s)). The sixth, *history*(s), is a sequence of generic operations. The initial state of the GKC is denoted by s_0. As in the generic controller, all sets are empty in s_0 except for create-requested, which is $\{T_0\}$. *history*(s_0) is,the empty sequence. As before, we define *returned*(s) = committed(s) $\cup$ aborted(s).

The transition relations for all operations except CREATE(T), where T is an access, are defined as for the generic controller, except that each operation π has an additional postcondition of the form $history(s) = history(s')\pi$. In other words, the history component of the state simply records the sequence of operations that have occurred. The transition relation for the CREATE(T) operation, where T is an access, is defined as follows.

- CREATE(T), T an access to X

  Pre: $T \in$ create-requested(s') $-$ created(s')
  $\forall T'$.COMMIT(T', v) visible-affects CREATE(T) in $history(s')$CREATE(T) $\Longrightarrow$
  INFORM-COMMIT-AT(X)OF(T') $\in history(s')$
  $\forall T'$.ABORT(T') prefix-affects CREATE(T) in $history(s')$CREATE(T) $\Longrightarrow$
  INFORM-ABORT-AT(X)OF(T') $\in history(s')$
  Post: created(s) = created(s') $\cup \{T\}$
  $history(s) = history(s')\pi$

At the point where an access to X is about to be created (and thus a lock test may occur), an explicit test is performed to verify that for every T' such that a COMMIT(T', v) or ABORT(T') operation affects the CREATE operation, the corresponding INFORM operation for T' has occurred at X.

6.1.2 Global Knowledge Systems

A Global Knowledge System is the composition of transactions, generic objects that satisfy Assumption 16 (explained below), and the Global Knowledge controller. Schedules of a Global Knowledge System are called Global Knowledge schedules.

The following lemma states the effects on Global Knowledge schedules of the precondition on CREATE operations in the Global Knowledge controller.

Lemma 11. Let $\alpha = \alpha'\pi$ be a Global Knowledge schedule, where $\pi = \text{CREATE}(T)$ and T is an access to X.

a. If π is visible-affected by $\text{COMMIT}(T', v)$ in α, for some transaction T', then $\text{INFORM-COMMIT-AT}(X)\text{OF}(T') \in \alpha'$.

b. If π is prefix-affected by $\text{ABORT}(T')$ in α, for some transaction T', then $\text{INFORM-ABORT-AT}(X)\text{OF}(T') \in \alpha'$.

Proof: Follows easily from the preconditions on CREATE in the definition of the Global Knowledge Controller. ■

The next lemma allows us to conclude that all results about generic schedules hold for Global Knowledge schedules as well.

Lemma 12. Every Global Knowledge schedule is a generic schedule.

Proof: It is not hard to see that this is true since the Global Knowledge Controller merely restricts the allowable schedules of the generic conroller and does not permit any new schedules. The full proof of this lemma would be analogous to Lemma 9 in [Herlihy *et al.* 1987]. ■

6.1.3 Global Knowledge Systems Guarantee Eager Diffusion

We proceed to prove that Global Knowledge Systems guarantee eager diffusion as follows. We define closure operators over sequences of generic operations based on the prefix-affects and visible-affects relations defined in Section 5.3, and prove that we can use these operators to extract from a generic schedule subsequences of generic operations that themselves comprise a generic schedule. In order that subsequences extracted using the visible-affects relation be generic schedules, we must make an assumption about the generic objects in the system (this is the assumption referred to in the previous section). Finally, we use the closure operators to prove that Global Knowledge Systems guarantee eager diffusion.

Definition 13. Let α be a sequence of generic operations and Π a set of events in α. Define *prefix-affects-close*(α, Π) to be the smallest subsequence β of α such that for every $\pi \in \Pi$, and for every $\phi \in \alpha$ such that $(\phi, \pi) \in$ *prefix-affects*(α), $\phi \in \beta$. Define *visible-affects-close*(α, Π) similarly, with *visible-affects*(α) substituted for *prefix-affects*(α).

A proof that the *prefix-affects-close* operator can used to extract generic schedules can be found in [Herlihy *et al.* 1987].

Lemma 14. Let α be a generic schedule, and Π a set of events in α. Then *prefix-affects-close*(α, Π) is a generic schedule.

Proof: Follows from Lemma 8 in [Herlihy *et al.* 1987]. ■

Now we want to prove that the *visible-affects-close* operator can be used to extract generic schedules. First, we prove a small lemma stating that a transaction can only be affected at an object by other transactions that are visible to it. Then we define our assumptions on objects, and use this along with the lemma to prove the result about *visible-affects-close*.

Lemma 15. Let α be a generic schedule. For all events ϕ, π in α such that *location*(ϕ) = an object X, and ϕ visible-affects π in α, T_ϕ is visible to T_π in α.

Proof: By induction on the length of visible-affects chains. In the base case, consider events ϕ and π such that ϕ directly-visible-affects π in α. If π is an output of X then, by definition of directly-visible-affects, T_ϕ is locally visible to T_π at X in α, and thus T_ϕ is visible to T_π in α. Otherwise, ϕ = REQUEST-COMMIT(T, v) for T an access to X and π = COMMIT(T, v), and clearly, T is visible to itself.

For the induction hypothesis, assume that the lemma holds for any appropriate ϕ and π with a visible-affects chain of length at most $k-1$. We will show that it holds for any ϕ and π with a visible-affects chain of length k.

Let ϕ and π be events in α such that *location*$(\phi) = X$, ϕ visible-affects π in α, and $\delta = \delta'\pi$ is a visible-affects chain between ϕ and π of length k. Let ψ be the last event in δ'. By the induction hypothesis, T_ϕ is visible to T_ψ in δ' and hence, in α. Now we consider cases for ψ.

1. ψ is an operation at an object X' and π is an output of X'. Then, by the induction hypothesis, T_ψ is visible to T_π in α. By transitivity of visibility, T_ϕ is visible to T_π.
2. ψ is an operation at a transaction T' and π is an output of T'. Then T_ψ is either the parent of T_π or a sibling of T_π. In either case, since T_ϕ is visible to T_ψ, it is also visible to T_π.
3. $T_\psi = T_\pi$. Then clearly, T_ϕ is visible to T_π in α.

■

The following assumption must be true of all objects in the system.[1]

Assumption 16. Let α be a schedule of a generic object X. Let Π be a set of events in α such that $\forall \pi \in \Pi . \forall T \in anc(T_\pi)$.INFORM-ABORT-AT$(X)OF(T) \notin \alpha$. Then *visible-affects-close*(α, Π) is a schedule of X.

[1] Though we have not proved it rigorously, we believe that the assumption is true of the locking objects in which we are primarily interested.

Lemma 17. Let α be a generic schedule, and Π a set of events in α such that $\forall \pi \in \Pi. \forall T \in anc(T_\pi).\text{ABORT}(T) \notin \alpha$. Then *visible-affects-close*(α, Π) is a generic schedule.

Proof: Let $\beta =$ *visible-affects-close*(α, Π). We must show that β is a schedule of the generic controller, and that for any non-access transaction T, $\beta \mid T$ is a schedule of T, and for any generic object X satisfying Assumption 16, $\beta \mid X$ is a schedule of X.

First, it is easy to see that β is a schedule of the generic controller. From the definition of the visible-affects relation, whenever β includes an output π of the controller, it will also include all previous operations required by the preconditions of the controller for π to occur.

Next, consider any non-access transaction T. From the definition of *visible-affects*(α) it is easy to see that either $\beta \mid T$ is a prefix of $\alpha \mid T$, possibly followed by some input operations at T, or $\beta \mid T = \Lambda$ (where Λ is the empty sequence). In either case, clearly $\beta \mid T$ is a schedule of T.

Finally, consider any generic object X that satisfies Assumption 16. We must show that $\beta \mid X$ is a schedule of X. If $\beta \mid X = \Lambda$ then clearly this is true. Otherwise, consider the set Φ of events in $\beta \mid X$. It is easy to see that $\beta \mid X =$ *visible-affects-close*$(\beta \mid X, \Phi)$. Now, if we can show that for each $\phi \in \Phi$, no INFORM-ABORT-AT(X) appears in $\beta \mid X$ for any ancestor of T_ϕ then Assumption 16 will tell us that $\beta \mid X$ is a schedule of X. For each $\phi \in \Phi$, either $\phi \in \Pi$ or ϕ visible-affects an event $\pi \in \Pi$. If $\phi \in \Pi$ then clearly T_ϕ is not a local orphan at X by the hypothesis of the lemma. So assume $\phi \notin \Pi$. Then there is some operation π in Π that is visible-affected by ϕ in β. By Lemma 15, T_ϕ is visible to T_π in β. Since T_π is not an orphan in α, no ancestor of π is aborted in β. Since T_ϕ is visible to T_π in β, no ancestor of T_ϕ below $lca(T_\phi, T_\pi)$ is aborted in β. Thus, no ancestor of T_ϕ is aborted in β. ■

Finally, we can prove that Global Knowledge Systems guarantee eager diffusion.

Theorem 18. Global Knowledge Systems guarantee eager diffusion.

Proof: Let α be any GKS schedule ending in $\pi = \text{CREATE}(T)$, where T is an access to X and T is not an orphan in α. Let U be an access to X. Assume $\alpha \vdash K_T(\mathcal{G}(U, T))$. We must prove $\alpha \vdash \mathcal{L}(U, T, X)$, that is, $\text{CREATE}(U) \in \alpha$ implies that either U is locally visible at X to T in α, or $\exists U' \in anc(U).\text{INFORM-ABORT-AT}(X)\text{OF}(U') \in \alpha$. Assume $\text{CREATE}(U) \in \alpha$, since otherwise the proof is trivial.

Consider $\beta =$ *prefix-affects-close*$(\alpha, \{\pi, \text{CREATE}(U)\})$. By Lemma 14, β is a generic schedule. Also, it is easy to see that $\beta \mid T = \text{CREATE}(T) = \alpha \mid T$. Since $\alpha \vdash K_T(\mathcal{G}(U, T))$, either $\text{ABORT}(U') \in \beta$ for $U' \in anc(U)$, or U is visible to T in β.

Suppose $\text{ABORT}(U') \in \beta$. Then $\text{ABORT}(U')$ occurs either in *prefix-affects-close*(α, π) or in *prefix-affects-close*$(\alpha, \text{CREATE}(U))$. If $\text{ABORT}(U')$ is in

prefix-affects-close(α, π), Lemma 11 tells us that INFORM-ABORT-AT(X)OF(U') is in α. Otherwise, if ABORT(U') is in *prefix-affects-close*$(\alpha,$ CREATE$(U))$ then Lemma 11 tells us that INFORM-ABORT-AT(X)OF(U') is in the prefix of α ending at CREATE(U), and thus is in α.

On the other hand, suppose no ancestor of U is aborted in β. Thus, U is visible to T in β. Then consider γ = *visible-affects-close*$(\alpha, \{\pi,$ CREATE$(U)\})$. By Lemma 17, γ is a generic schedule. It is easy to see that γ is a subsequence of β such that $\gamma \mid T = \alpha \mid T$. Since $\alpha \vdash K_T(\mathcal{G}(U,T))$, we must have U visible to T in γ (as no ancestor of U is aborted in γ). But the COMMIT events for the ancestors of U that are proper descendants of $lca(U,T)$ cannot be in the visible-affects-closure of CREATE(U), so they must be in the visible-affects-closure of π. Lemma 11 assures us that INFORM-COMMIT-AT(X) operations have occurred in α for each of these ancestors of U. Thus, U is locally visible to T in α, as required. ■

An additional result, which we will use later in proving that Inference Optimized Systems guarantee eager diffusion, is that any system that implements a Global Knowledge System also guarantees eager diffusion.

Theorem 19. Let S be an implementation of a Global Knowledge System. Then S guarantees eager diffusion.

Proof: If S is an implementation of a Global Knowledge System, then every schedule of S is a GKS schedule. We just proved in Theorem 18 that all GKS schedules have the property required for eager diffusion. ■

6.2 Local Unoptimized Systems

As an intermediate step between the Global Knowledge System and the system that models our simplified protocol (including inferences) we define a system that models the way in which local commit and abort information can be transferred through the transaction tree, using explicitly recorded information. A Local Unoptimized (LU) System models this flow of commit and abort information. It is composed of transactions, generic objects, and the Local Unoptimized Controller (LUC). What we will prove in the main theorem of this section is that the Local Unoptimized Controller simulates the Global Knowledge Controller in that all well-formed schedules of the LUC are also schedules of the GKC. This implies that, given an LU System and a GK System with the same transactions and objects, all schedules of the LU System are schedules of the GK System. In particular then, no transaction can tell whether it is in an LU System or a GK System since anything it could see in the first system, it could see in the second.

6.2.1 Local Unoptimized Controller

The Local Unoptimized Controller (LUC) has the same seven operations as the generic controller. Each state s of the LUC consists of nine components. The first five are the same as for the generic controller (*i.e.*, create-requested(s), created(s), commit-requested(s), committed(s), and aborted(s)). The sixth and seventh, informed-commit(s) and informed-abort(s), are mappings from objects to sets of transactions. These components record the transaction commits and aborts, respectively, of which each object has been informed. The eight and ninth components, abort-map(s) and commit-map(s), are mappings from operations to sets of transactions. They record the transactions whose aborts and commits, respectively, affect each operation. We allow the sets abort-map(s)(π) and commit-map(s)(π) to include more transactions than those whose aborts or commits affect π in an execution.

As usual, the initial state is denoted by s_0, and all sets are initially empty in s_0 except for create-requested, which is $\{T_0\}$. The functions commit-map(s_0) and abort-map(s_0) map each operation to the empty set. As usual, define *returned*(s) = committed(s) $\cup$ aborted(s).

The transition relations for the operations of the LUC are defined as follows. As usual, s' indicates the state before the indicated operation, and s indicates the state after the operation. Also, every operation π contains the following additional postconditions:

1. $\forall \phi.$ commit-map(s')(ϕ) $\subseteq$ commit-map(s)(ϕ)
2. $\forall \phi.$ abort-map(s')(ϕ) $\subseteq$ abort-map(s)(ϕ)
3. $\forall \phi.$ abort-map(s)(ϕ) $\subseteq$ aborted(s)
4. $\forall \phi.$ commit-map(s)(ϕ) $\subseteq$ committed(s)

The first two conditions state that all commit-maps and abort-maps only grow over the course of an execution. The third and fourth postconditions constrain commit-maps and abort-maps to contain committed and aborted transactions, respectively. This limits the nondeterminism of the controller.

All other state components remain the same from s' to s unless explicitly mentioned in the postconditions below.

- π = REQUEST-CREATE(T)

  Post: create-requested(s) = create-requested(s') $\cup \{T\}$
  $\forall \phi. location(\phi) = location(\pi) \Longrightarrow$
  commit-map(s')(ϕ) $\subseteq$ commit-map(s)(π) and
  abort-map(s')(ϕ) $\subseteq$ abort-map(s)(π)

- π = REQUEST-COMMIT(T, v), where T is a non-access transaction

Post: commit-requested(s) = commit-requested$(s') \cup \{(T, v)\}$
$\forall \phi.$*location*(ϕ) = *location*$(\pi) \Longrightarrow$
commit-map$(s')(\phi) \subseteq$ commit-map$(s)(\pi)$ and
abort-map$(s')(\phi) \subseteq$ abort-map$(s)(\pi)$

- π = REQUEST-COMMIT(T, v), where T is an access to an object X

Post: commit-requested(s) = commit-requested$(s') \cup \{(T, v)\}$
$\forall \phi.$*location*(ϕ) = *location*$(\pi) \Longrightarrow$
visible(*transaction*$(\phi), T, X, s') \Longrightarrow$ commit-map$(s')(\phi) \subseteq$ commit-map$(s)(\pi)$
and abort-map$(s')(\phi) \subseteq$ abort-map$(s)(\pi)$
where *visible*(T', T, X, s) =
$\forall T''$ ancestor of T' and proper descendant of lca(T, T').
$T'' \in$ informed-commit$(s)(X)$

- π = CREATE(T)

Pre: $T \in$ create-requested$(s') -$ created(s')
if T is an access to object X then
commit-map(s')(REQUEST-CREATE$(T)) \subseteq$ informed-commit$(s')(X)$ and
abort-map(s')(REQUEST-CREATE$(T)) \subseteq$ informed-abort$(s')(X)$
Post: created(s) = created$(s') \cup \{T\}$
commit-map(s')(REQUEST-CREATE$(T)) \subseteq$ commit-map$(s)(\pi)$
abort-map(s')(REQUEST-CREATE$(T)) \subseteq$ abort-map$(s)(\pi)$

- π = COMMIT(T, v)

Pre: $(T, v) \in$ commit-requested(s')
$T \notin$ *returned*(s')
Post: committed(s) = committed$(s') \cup \{T\}$
commit-map(s')(REQUEST-COMMIT$(T, v)) \subseteq$ commit-map$(s)(\pi)$
abort-map(s')(REQUEST-COMMIT$(T, v)) \subseteq$ abort-map$(s)(\pi)$
$T \in$ commit-map$(s)(\pi)$

- π = ABORT(T)

Pre: $T \in$ create-requested$(s') -$ *returned*(s')
Post: aborted(s) = aborted$(s') \cup \{T\}$
commit-map(s')(REQUEST-CREATE$(T)) \subseteq$ commit-map$(s)(\pi)$
abort-map(s')(REQUEST-CREATE$(T)) \subseteq$ abort-map$(s)(\pi)$
$T \in$ abort-map$(s)(\pi)$

- π = INFORM-COMMIT-AT(X)OF(T)[2]

  Pre: $T \in$ committed(s')
  Post: $\forall v.(T,v) \in$ commit-requested(s') $\Longrightarrow$
  commit-map(s')(COMMIT(T,v)) $\subseteq$ commit-map(s)(π) and
  abort-map(s')(COMMIT(T,v)) $\subseteq$ abort-map(s)(π)
  informed-commit(s)(X) = informed-commit(s')(X) $\cup$ $\{T\}$

- π = INFORM-ABORT-AT(X)OF(T)

  Pre: $T \in$ aborted(s')
  Post: commit-map(s')(ABORT(T)) $\subseteq$ commit-map(s)(π)
  abort-map(s')(ABORT(T) $\subseteq$ abort-map(s)(π)
  informed-abort(s)(X) = informed-abort(s')(X) $\cup$ $\{T\}$

Essentially, this controller is passing around information in commit-maps and abort-maps based on the affects relations defined earlier. Whenever an operation ϕ directly-visible-affects another operation π, we make sure to include ϕ's commit-map in π's commit-map. Likewise, whenever ϕ directly-prefix-affects π, we include ϕ's abort-map in π's abort map. By the time we get to a CREATE for an access, we expect all transactions whose COMMITs and ABORTs affect the CREATE (according to the appropriate affects relation) to be in the commit-map or abort-map for the preceding REQUEST-CREATE and thus "locally" available to the CREATE. Below we prove that this is, in fact, the case.

6.2.2 Local Unoptimized Systems

Local Unoptimized Systems are composed of transactions, generic objects that satisfy Assumption 16, and the Local Unoptimized Controller. Schedules of a Local Unoptimized System are called Local Unoptimized schedules.

A fact easily ascertained by examining the LUC program is:

Lemma 20. *All sets of transactions in the LUC state increase monotonically.*

The following two lemmas state properties about the transfer of commit and abort information between operations related by the affects relations.

Lemma 21. Let α be an LUC schedule, γ an LUC execution such that *schedule*(γ) = α, ϕ and π events in α, and s' and s the states in γ immediately after ϕ and π, respectively. Then

[2]The postcondition here may be confusing. Since the value returned in the COMMIT operation is not part of the INFORM-COMMIT the postcondition requires commit-map(s')(COMMIT(T,v)) to be included for all (T,v) in commit-requested(s'). For a given T, however, only one COMMIT(T,v) will occur, so there will be only one such pair in commit-requested(s').

(1) ϕ directly-visible-affects π in $\alpha \Longrightarrow$ commit-map(s')(ϕ) $\subseteq$ commit-map(s)(π) .

(2) ϕ directly-prefix-affects π in $\alpha \Longrightarrow$ abort-map(s')(ϕ) $\subseteq$ abort-map(s)(π) .

Proof: Both (1) and (2) follow directly from the definition of the LUC and Lemma 20. ■

Lemma 22. Let α be an LUC schedule, and γ an LUC execution such that *schedule*(γ) = α.

(1) For all ϕ, π events in α such that ϕ visible-affects π in α, s' and s states in γ after ϕ and π, respectively, commit-map(s')(ϕ) $\subseteq$ commit-map(s)(π).

(2) For all ϕ, π events in α such that ϕ prefix-affects π in α, s' and s states in γ after ϕ and π, respectively, abort-map(s')(ϕ) $\subseteq$ abort-map(s)(π).

Proof: (1) By induction on the number of events in a visible-affects chain between ϕ and π. The basis is when the number of events is 0. Then ϕ directly-visible-affects π and the claim follows from Lemma 21.

For the induction hypothesis, assume that the claim is true for ϕ and π where a visible-affects chain contains at most $k-1$ events between ϕ and π. We will show that it is true for all ϕ, π with visible-affects chains containing k operations between ϕ and π. Fix any ϕ, π and a visible-affects chain with k events between ϕ and π (there could be more than one). Let ψ be the event in the chain immediately preceding ϕ (*i.e.*, ψ directly-visible-affects π in α). Obviously, ϕ visible-affects ψ and there is a visible-affects chain between ϕ and ψ containing at most $k-1$ events (it is just the same chain with π eliminated). Let s'' be the state in γ immediately after ψ. By the induction hypothesis, commit-map(s')(ϕ) $\subseteq$ commit-map(s'')(ψ). By Lemma 21, commit-map(s'')(ψ) $\subseteq$ commit-map(s)(π). Thus by transitivity, commit-map(s')(ϕ) $\subseteq$ commit-map(s)(π).

(2) The same argument applies, substituting prefix-affects for visible-affects and abort-map for commit-map. ■

The following key lemma says that any time the commit (or abort) of a transaction T visible-affects (or prefix-affects) another operation ϕ in some schedule of the LU system, then T will be in the commit-map (or abort-map) of ϕ after ϕ occurs. This lemma makes the proof of the first simulation theorem straightforward.

Lemma 23. Let α be an LUC schedule, π an operation in α, and s a state of the LUC after α. Then

(1) $\forall T$.COMMIT(T, v) visible-affects an instance of π in $\alpha \Longrightarrow T \in$ commit-map(s)(π) .

(2) $\forall T.\text{ABORT}(T)$ prefix-affects an instance of π in $\alpha \Longrightarrow T \in \text{abort-map}(s)(\pi)$.

Proof: (1) Let $\psi = \text{COMMIT}(T, v)$ be an event in α. Let ϕ be an instance of π in α such that ψ visible-affects ϕ. Let γ be an LUC execution containing s such that $schedule(\gamma) = \alpha$. Let s' and s'' be the states in γ after ϕ and ψ, respectively. By Lemma 22, commit-map$(s'')(\psi) \subseteq$ commit-map$(s')(\phi)$. Looking at the LUC postcondition on ψ, we see that $T \in$ commit-map$(s'')(\psi)$. Thus $T \in$ commit-map$(s')(\pi)$ also. Then by Lemma 20, $T \in$ commit-map$(s)(\pi)$.

(2) The same argument applies with $\psi = \text{ABORT}(T)$, prefix-affects replacing visible-affects, and abort-map replacing commit-map. ■

6.2.3 Simulation of GK Systems by LU Systems

The theorem of this section states that the LUC simulates the GKC in the strong sense that every well-formed schedule of the first controller is a schedule of the second. A simple corollary is that, given LU and GK systems composed of the same transactions and generic objects, every schedule of the LU System is also a schedule of the GK System.

Theorem 24. Every well-formed LUC schedule is a GKC schedule.

Proof: By induction on the length of LUC schedules. Let α be an LUC schedule. The basis, α of length 0, is trivial. For the induction hypothesis, assume the claim holds for α of length $n-1$. We will show that it holds for α of length n.

Let $\alpha = \alpha'\pi$ be an LUC schedule of length n where π is a single operation. Let s' be a state of the LUC after α' in which π is enabled, and let t' be the state of the GKC after α'. (By the induction hypothesis, α' is a schedule of the GKC and so t' exists. Since the GKC is deterministic, t' is uniquely defined.) We must show that π is enabled in state t'. The only case that is not immediate is where $\pi = \text{CREATE}(T)$, where T is an access to X.[3] What we must show, in effect, is:

(1) $\forall T'.\text{COMMIT}(T', v)$ visible-affects $\text{CREATE}(T)$ in $\alpha \Longrightarrow$
INFORM-COMMIT-AT(X)OF$(T') \in \alpha$, and
(2) $\forall T'.\text{ABORT}(T')$ prefix-affects $\text{CREATE}(T)$ in $\alpha \Longrightarrow$
INFORM-ABORT-AT(X)OF$(T') \in \alpha$.

(1) Let T' be such that $\text{COMMIT}(T', v)$ visible-affects $\text{CREATE}(T)$ in α. By well-formedness conditions and the definition of the visible-affects relation, we know that there must be a REQUEST-CREATE(T) operation in α' and that

[3] It is easy to see that the first five state components of s' will be equal to the corresponding components of t'. Since the GKC preconditions for operations other than CREATE only mention those components it is easy to see that when $\pi \neq$ CREATE, π will be enabled in t' if it is enabled in s'.

COMMIT(T', v) visible-affects REQUEST-CREATE(T) in α'. By Lemma 23, $T' \in$ commit-map(s')(REQUEST-CREATE(T)). Since π is enabled in s', it must also be the case that $T' \in$ informed-commit$(s')(X)$. Looking at the LUC program we see that the informed-commit set is only added to by the INFORM-COMMIT operation. Thus INFORM-COMMIT-AT(X)OF(T) must occur in α'.

(2) A similar argument applies (using the prefix-affects relation instead of visible-affects). ■

6.3 Inference Optimized Systems

The final step towards modelling the simplified protocol is to verify that the inference mechanisms that reduce the size of the committed set are valid. In the LUC, every time a transaction commits it is added to a commit-map. In the simplified protocol, we do not actually remember all commits since some commits can be inferred based on information in the aborts set and information about the transactions (the structure of the transaction tree and which transactions have been created).

In this section we define the Inference Optimized Controller (IOC). The controller uses explicitly recorded information about commits, together with information about aborts and the structure of transactions, to test that all relevant inform operations have occurred before an access to an object is enabled. We define Inference Optimized Systems and show that the Inference Optimized Controller simulates the Local Unoptimized Controller in the sense that all well-formed schedules of the IOC are schedules of the LUC. This implies that every well-formed IOC schedule is also a GKC schedule, and thus, every Inference Optimized schedule is a Global Knowledge schedule. The proof that IO Systems are correct is then trivial.

6.3.1 Assumptions About Transactions

The definition of the IOC relies on some assumptions about the structure of transactions.

We assume that the children of a transaction T are partitioned into *concurrency groups*. The concurrency groups are partially ordered by the relation $<$ such that a child T' of T in concurrency groups G is only request-created if, for all groups G' of transactions that have already been request-created by T, $G' \leq G$. All transactions must obey this restriction. A transaction may request the creation of any of the transactions in a particular currency group G without waiting for returns of other transactions in G whose creates have been requested. A transaction may not request the creation of any member of a group until all transactions in other groups whose creates have been requested have returned.

A child T' of T in group G is a *later sequential sibling* of another child T'' of T in group G' if $G' < G$. Similarly, T'' is a *prior sequential sibling* of T' in that case.

Children of T in the same concurrency group are called *concurrent siblings.*

Every transaction may itself be classified as *concurrent* or *sequential.* A transaction is sequential if its concurrency group is of size 1. Otherwise the transaction is concurrent. The predicate *concurrent*(T) is true if and only if transaction T is concurrent.

6.3.2 Inference Optimized Controller

Each state of the Inference Optimized Controller has the same first eight state components as the LUC. Instead of the commit-map, the ninth component of the IOC state is called cmap(s). It is also a mapping from operations to sets of transactions. However, it is named differently to emphasize that it will contain different information than the commit-map of the LUC. Rather then recording all commits known to a particular operation, it records only those commits that cannot be inferred by other means.

The following notation and definitions are used in the description of the Inference Optimized Controller.

We introduce a shorthand for referring to relationships of transactions in the transaction tree. We use the the notations *desc*(T), *prop-desc*(T), *anc*(T), and *prop-anc*(T) to refer to the sets of descendants, proper descendants, ancestors, and proper ancestors, respectively, of a transaction T. For a transaction T, and a set of transactions S, define:

$cpa(T, S) = \{T' \in S \mid parent(T') \in prop\text{-}anc(T)\}$
("children of proper ancestors of T in S")

$ca(T, S) = \{T' \in S \mid parent(T') \in anc(T)\}$
("children of ancestors of T in S")

$c(T, S) = \{T' \in S \mid parent(T') = T\}$
("children of T in S")

$cpd(T, S) = \{T' \in S \mid parent(T') \in prop\text{-}desc(T)\}$
("children of proper descendants of T in S")

$cd(T, S) = \{T' \in S \mid parent(T') \in desc(T)\}$
("children of descendants of T in S")

The notation T_π refers to the transaction argument of operation π. For example, if π = REQUEST-COMMIT(T, v), then $T_\pi = T$, and if π = COMMIT(T', v) then $T_\pi = T'$. Note that in the first case, $location(\pi) = T_\pi$, while in the second case $location(\pi) = parent(T_\pi)$.

Given an operation and a state of the system in which the operation has occurred, it is possible, by reasoning about sequentiality of transactions, to infer that particular

transactions must have returned in the state if they were ever created. *seq-infer*(s, π) is a set of transactions that may be inferred to have returned in this way. We restrict the set to contain only children of ancestors of T_π because they are all that will be needed for our purposes. (Later we will apply a closure operation to the set to obtain the descendants of the transactions in the set that have actually committed).

seq-infer(s, π) =
  if *has-occurred*(s, π) then
    $\{T \in$ created$(s) \mid T_\pi$ is a descendant of a later sequential sibling of T, or
      $\pi =$ REQUEST-COMMIT(T', v) or COMMIT(T', v)
      or INFORM-COMMIT-AT(X)OF(T') and T is a child of $T'\}$
  else $\emptyset$

has-occurred(s, π) =
  if $\pi =$ REQUEST-CREATE(T) then $T \in$ create-requested(s)
  elseif $\pi =$ REQUEST-COMMIT(T, v) then $(T, v) \in$ commit-requested(s)
  elseif $\pi =$ CREATE(T) then $T \in$ created(s)
  elseif $\pi =$ COMMIT(T, v) then $T \in$ committed(s)
  elseif $\pi =$ ABORT(T) then $T \in$ aborted(s)
  elseif $\pi =$ INFORM-COMMIT-AT(X)OF(T) then $T \in$ informed-commit$(s)(X)$
  elseif $\pi =$ INFORM-ABORT-AT(X)OF(T) then $T \in$ informed-abort$(s)(X)$

A complementary definition to *seq-infer*, *conc-infer*(s, π) is the set of transactions concurrent with T_π that are known to be committed by operation π in state s. The definition does not parallel *seq-infer* in that information about concurrent transactions must be recorded explicitly because it cannot be inferred. Again, we are interested only in children of proper ancestors of T_π. *conc-with*(T, S) is the subset of transactions in S having ancestors that are concurrent siblings of ancestors of T.

conc-with$(T, S) = \{T' \in S \mid T, T'$ are descendants of concurrent siblings $\}$

conc-infer$(s, \pi) =$ *conc-with*$(T_\pi,$ *cpa*$(T_\pi,$ cmap$(s)(\pi)))$

cmt-closure is a closure operation on a set of created transactions C, a set of aborted transactions A, and a set of of returned actions, R, that produces all created transactions that are descendants of transactions in R and that have committed.

$desc(R) = \bigcup_{T \in R} desc(T)$

$non\text{-}orphans(A, R) = \{T \in desc(R) \mid anc(T) \cap desc(R) \cap A = \emptyset\}$

$cmt\text{-}closure(C, A, R) = C \cap desc(R) \cap non\text{-}orphans(A, R)$

The *cmt-closure*s that we will be interested in for the IOC are those starting from *seq-infer* and *conc-infer*.

$$\textit{all-seq}(s,\pi) = \textit{cmt-closure}(\text{created}(s), \text{abort-map}(s)(\pi), \textit{seq-infer}(s,\pi))$$

$$\textit{all-conc}(s,\pi) = \textit{cmt-closure}(\text{created}(s), \text{abort-map}(s)(\pi), \textit{conc-infer}(s,\pi))$$

$$\textit{known-vis}(s,\pi) = \textit{all-seq}(s,\pi) \cup \textit{all-conc}(s,\pi)$$

We now proceed with the definition of the Inference Optimized Controller. As usual, the initial state of the IOC is denoted by s_0, and all sets are initially empty in s_0 except for create-requested, which is $\{T_0\}$. The functions cmap(s_0) and abort-map(s_0) map each operation to the empty set. As usual, define $\textit{returned}(s) = \text{committed}(s) \cup \text{aborted}(s)$.

In the transition relations for the operations of the IOC defined below, every operation π contains the following additional postconditions:

1. $\text{cmap}(s')(\pi) \subseteq \text{cmap}(s)(\pi)$
2. $\text{cmap}(s)(\pi) \subseteq \text{committed}(s)$
3. $\text{abort-map}(s')(\pi) \subseteq \text{abort-map}(s)(\pi)$
4. $\text{abort-map}(s)(\pi) \subseteq \text{aborted}(s)$
5. $\forall T \in \text{cmap}(s)(\pi). \forall T' \in \text{aborted}(s) \cap \textit{prop-desc}(T).$
   $\textit{anc}(T') \cap \text{abort-map}(s)(\pi) \cap \textit{prop-desc}(T) \neq \emptyset$

The first four postconditions are similar to those in the LUC, although they apply only to π, whereas in the LUC they are quantified over all operations. The fifth postcondition will be explained after we describe the controller.

All other state components remain the same from s' to s unless explicitly mentioned in the postconditions below.

- $\pi = \text{REQUEST-CREATE}(T)$

  Post: $\text{create-requested}(s) = \text{create-requested}(s') \cup \{T\}$
  $\forall \phi. \textit{location}(\phi) = \textit{location}(\pi) \Longrightarrow$
  $\textit{conc-with}(T, \textit{cpa}(T, \text{cmap}(s')(\phi))) \subseteq \text{cmap}(s)(\pi)$ and
  $\text{abort-map}(s')(\phi) \subseteq \text{abort-map}(s)(\pi)$

- $\pi = \text{REQUEST-COMMIT}(T, v)$

  Post: $\text{commit-requested}(s) = \text{commit-requested}(s') \cup \{(T, v)\}$
  $\forall \phi. \textit{location}(\phi) = \textit{location}(\pi) \Longrightarrow$
  $\textit{conc-with}(T, \textit{cpa}(T, \text{cmap}(s')(\phi))) \subseteq \text{cmap}(s)(\pi)$ and
  $\text{abort-map}(s')(\phi) \subseteq \text{abort-map}(s)(\pi)$

- $\pi = \text{CREATE}(T)$

  Pre: $T \in \text{create-requested}(s') - \text{created}(s')$
  if T is an access to object X then
  $\mathit{known\text{-}vis}(s', \text{REQUEST-CREATE}(T)) \subseteq \text{informed-commit}(s')(X)$ and
  $\text{abort-map}(s')(\text{REQUEST-CREATE}(T)) \subseteq \text{informed-abort}(s')(X)$
  Post: $\text{created}(s) = \text{created}(s') \cup \{T\}$
  $\text{cmap}(s')(\text{REQUEST-CREATE}(T)) \subseteq \text{cmap}(s)(\pi)$
  $\text{abort-map}(s')(\text{REQUEST-CREATE}(T)) \subseteq \text{abort-map}(s)(\pi)$

- $\pi = \text{COMMIT}(T, v)$

  Pre: $(T, v) \in \text{commit-requested}(s')$
  $T \notin \mathit{returned}(s')$
  Post: $\text{committed}(s) = \text{committed}(s') \cup \{T\}$
  $\text{cmap}(s')(\text{REQUEST-COMMIT}(T, v)) \subseteq \text{cmap}(s)(\pi)$
  $\text{abort-map}(s')(\text{REQUEST-COMMIT}(T, v)) \subseteq \text{abort-map}(s)(\pi)$
  if $\mathit{concurrent}(T)$ then $T \in \text{cmap}(s)(\pi)$

- $\pi = \text{ABORT}(T)$

  Pre: $T \in \text{create-requested}(s') - \mathit{returned}(s')$
  Post: $\text{aborted}(s) = \text{aborted}(s') \cup \{T\}$
  $\text{cmap}(s')(\text{REQUEST-CREATE}(T)) \subseteq \text{cmap}(s)(\pi)$
  $\text{abort-map}(s')(\text{REQUEST-CREATE}(T)) \cup \{T\} \subseteq \text{abort-map}(s)(\pi)$

- $\pi = \text{INFORM-COMMIT-AT}(X)\text{OF}(T)$

  Pre: $T \in \text{committed}(s')$
  Post: $\forall v.(T, v) \in \text{commit-requested}(s') \Longrightarrow$
  $\text{cmap}(s')(\text{COMMIT}(T, v)) \subseteq \text{cmap}(s)(\pi)$ and
  $\text{abort-map}(s')(\text{COMMIT}(T, v)) \subseteq \text{abort-map}(s)(\pi)$
  $\text{informed-commit}(s)(X) = \text{informed-commit}(s')(X) \cup \{T\}$

- $\pi = \text{INFORM-ABORT-AT}(X)\text{OF}(T)$

  Pre: $T \in \text{aborted}(s')$
  Post: $\text{cmap}(s)(\text{ABORT}(T)) \subseteq \text{cmap}(s)(\pi)$
  $\text{abort-map}(s')(\text{ABORT}(T)) \subseteq \text{abort-map}(s)(\pi)$
  $\text{informed-abort}(s)(X) = \text{informed-abort}(s')(X) \cup \{T\}$

This controller incorporates the two inference mechanisms described in Chapter 3 and used in our simplified protocol. Sequential inference is intended to be captured in the definition of *seq-infer*. It says that if a transaction T is running then any transaction T' that is a descendant of a prior sequential sibling of an ancestor of T and that has been created must have finished. Descendant inference is captured by the definition of *cmt-closure*, which says that if T knows that T' has finished then any descendants of T' that are not in T's abort-map must have committed. One way T could know that T' is finished is by sequential inference, and the other is by T''s presence in T's cmap.

In our informal description of the protocol the inference mechanism was used in the lock propagation rules. The information in the committed and aborted sets, as well as inferences that could be made from that information, was used to determine whether a current lock holder had aborted or committed up to its least common ancestor with a lock requestor. In our formal model we do not talk about the specific actions taken by an object that allow an access to occur (and we do not assume that objects use locking for concurrency control). So we must model the inferences in a different way. In the IOC, before an access to an object is created we require INFORM operations to occur at the object for all transactions that could be inferred to have committed or aborted based on the access operation's cmap and abort-map. Thus we are "precomputing" all inferences that an object could possibly need to make for an access rather than computing only the specific inferences that are actually needed for the access. If we had more information about the objects in the system we could limit the INFORM operations required at an object for an access to occur to only those actually used for the access.

This controller differs in detail from the LUC in a number of ways. The major differences involve the information added to cmaps, as compared to the information added to commit-maps in the LUC, and the precondition on CREATE(s) of accesses that delays the CREATE until all necessary INFORM operations have occurred.

One difference in the information added to cmaps can be seen in the postcondition for COMMIT. In the IOC we only add a transaction to a cmap if it is concurrent, as compared to the LUC where we always add transactions to commit-maps when they commit. In the IOC, if a transaction is sequential, all later transactions will be able to infer that it has committed.

Another difference in the information added to cmaps shows up in the postconditions for REQUEST-CREATE and REQUEST-COMMIT for a transaction T. For π = REQUEST-CREATE or REQUEST-COMMIT, instead of passing ϕ's cmap to π for all operations ϕ that directly-visible-affect π (as is done in the LUC), we pass through only part of ϕ's cmap, namely, the children of proper ancestors of T that are concurrent with T. The COMMITs of transactions that are sequential with T will be inferred through sequential inferences. Those that are descendants of the concurrent children of proper ancestors of T will be inferred by the *cmt-closure* over the concurrent infer-

ences. Any transactions other than children of proper ancestors of T that happen to be in ϕ's cmap must be extraneous to T. For example, consider a transaction T' that is in ϕ's cmap and that is not a descendant of any transaction in the cmap that is a child of a proper ancestor of T. If T does not know about the commit of T''s ancestor that is a child of a proper ancestor of T, then it cannot really know about T''s commit (and knowing it would not be sufficient for T to get a lock from T', anyway).

The third difference between the IOC and the LUC is in the precondition on CREATEs of accesses. The IOC requires inform operations for all transactions that can be inferred to have committed, while the LUC requires inform operations for transactions in the commit-map. We will show in the proof that *known-vis* effectively plays the same role in the IOC that commit-map plays in the LUC.

There are two other technical differences. The first is that the postconditions on all operations for the LUC allow commit-maps and abort-maps to change for all ϕ at each step of an execution. In the IOC, the cmap and abort-map for an operation may change only when the operation occurs. (If there are multiple instances of the operation then the sets may change multiple times. The only operations that may have multiple instances in a schedule of the system are the INFORM operations.)

The second technical difference is the addition of the fifth postcondition on all operations in the IOC. The postcondition has the effect of limiting the amount of non-determinism in the controller. It requires that whenever a transaction is added to a cmap, enough information is added to the corresponding abort-map to infer which of the descendants of the transaction are committed. If we allow a transaction with aborted descendants to be added to the cmap without adding corresponding information to the abort-map then we might incorrecty infer that those descendants are committed. In an implementation of the controller that keeps only the minimal amount of information in cmaps this postcondition would be redundant. (This claim requires a proof, which should not be very difficult, although we do not provide one here.)

6.3.3 Inference Optimized Systems

Inference Optimized Systems are composed of transactions, generic objects that satisfy Assumption 16, and the Inference Optimized Controller.

The following lemmas about the behavior of the IOC follow easily from the IOC program and are offered without proof. For these lemmas, let α be an IOC schedule, and γ an IOC execution such that $schedule(\gamma) = \alpha$.

Lemma 25. Let $\phi = \text{COMMIT}(T, v) \in \alpha$ and s a state some time after ϕ in γ. Then $T \in \text{committed}(s)$.

Lemma 26. Let ϕ be any operation and s a state in γ. Then $\text{cmap}(s)(\phi) \subseteq \text{committed}(s)$.

Lemma 27. Let ϕ be any operation and s a state in γ. Then *conc-infer*$(s, \phi) \subseteq$ committed(s).

Lemma 28. Let s be a state of the IOC during α and $T \in$ committed(s). Then Then COMMIT(T, v) occurs before s in α.

Lemma 29. Let ϕ be any operation and s a state some time after an instance of ϕ in γ. Let ϕ' be the last instance of ϕ before s and s' the state immediately after ϕ' in γ. Then cmap$(s')(\phi) =$ cmap$(s)(\phi)$ and abort-map$(s')(\phi) =$ abort-map$(s)(\phi)$.

Lemma 30. Let $\phi =$ ABORT$(T) \in \alpha$ and s a state in γ some time after ϕ. Then $T \in$ abort-map$(s)(\phi)$.

Lemma 31. All sets of transactions in the IOC state increase monotonically.

Lemma 32. Let s' and s be states in γ. Then committed$(s') \cap$ aborted$(s) = \emptyset$.

Lemma 33. Let $\phi =$ COMMIT$(T, v) \in \alpha$, where T is a concurrent transaction, and s a state some time after ϕ in γ. Then $T \in$ cmap$(s)(\phi)$.

6.3.4 Simulation of LU Systems by IO Systems

What we would now like to prove is that the IOC simulates the LUC in the strong sense that every well-formed schedule of the IOC is a schedule of the LUC. The proof here is much more subtle than the previous simulation proof (and therefore requires *many more lemmas*).

To show that every well-formed schedule of the IOC is a schedule of the LUC we will choose a state mapping relating states of the IOC to states of the LUC and show that, given any well-formed execution of the IOC, we can apply the state mapping to obtain an execution of the LUC.[4] The proof is inductive. For each step (s', π, s) taken by the IOC in the execution, we will show that there is a step (t', π, t) of the LUC such that s' is mapped to t' and s is mapped to t.

We map a state s of the IOC to a state t of the LUC as follows. Every component of t except commit-map is equal to the corresponding component of s. For every operation ϕ, we let commit-map$(t)(\phi)$ be *known-vis*(s, ϕ). The motivation for the choice of mapping derives from the definition of the IOC. The IOC precondition on CREATEs of accesses requires *known-vis* to be a subset of informed-commit while the LUC precondition on the operation requires commit-map to be a subset of informed-commit. All state components of the IOC, except cmap, are used in the same manner as in the LUC. Thus it will be easy, under this state mapping, to show that any operation enabled in a state of the IOC is enabled in the corresponding state of the LUC. The hard part is to show that the appropriate LUC postconditions involving commit-map hold under the state mapping.

[4] Methods for proving these types of simulations are presented in more detail in [Lynch & Tuttle 1987].

Our plan of attack is as follows. First we present the straightforward proof that any operation that is enabled in a state of the IOC is enabled in the corresponding state of the LUC. Then we attack the hard part, proving a lemma (Lemma 52) that allows us to show that the LUC postconditions involving commit-map for each individual operation are satisfied under the state mapping. Then we prove that the two LUC postconditions on all operations involving commit-map for each individual operation are satisfied under the state mapping. Finally, we pull the results together in Theorem 57 to prove that the IOC strongly simulates the LUC.

The following lemma is used in the proof of the simulation theorem to show that if an operation is enabled in a state of the IOC then it is enabled in a corresponding state of the LUC.

Lemma 34. Let s be a state of the IOC, π any operation enabled in state s, and t a state of the LUC that satisfies the following state mapping:

$\forall \phi.$ commit-map$(t)(\phi) =$ *known-vis*(s, ϕ), and
Every other component of t = the corresponding component of s.

Then π is enabled in state t.

Proof: By case analysis. We must show that the LUC preconditions are satisfied in state t for each output operation. The only interesting case is when $\pi =$ CREATE(T) where T is an access to an object X and the interesting precondition is:

commit-map(t)(REQUEST-CREATE$(T)) \subseteq$ informed-commit$(t)(X)$, and
abort-map(t)(REQUEST-CREATE$(T)) \subseteq$ informed-abort$(t)(X)$.

From the IOC preconditions for π we know that:

known-vis$(s,$ REQUEST-CREATE$(T)) \subseteq$ informed-commit$(s)(X)$, and
abort-map(s)(REQUEST-CREATE$(T)) \subseteq$ informed-abort$(s)(X)$.

From the state mapping, we know:
commit-map(t)(REQUEST-CREATE$(T)) =$ *known-vis*$(s,$ REQUEST-CREATE$(T))$,
and all other components of t are equal to the corresponding components of s. Using the state mapping to substitute into the preconditions for the IOC, the result follows directly.

For all other cases of output operations π, the result follows directly from the definition of the IOC and the state mapping. ■

Now we move on to the more difficult part of the proof. Lemma 52 states a monotonicity property for the inferences about commits represented by *known-vis*. It says that we can make at least the same inferences for an operation π when it occurs that we could make for an operation that directly-visible-affects π. Recall that the LUC passes around commit-maps according to the directly-visible-affects relation:

in an LUC execution, whenever an earlier operation directly-visible-affects a later operation, the commit-map for the earlier operation is included in the commit-map for the later operation. Lemma 52 shows that, in effect, the IOC "passes around" the inferences represented by *known-vis* in the same manner. This is exactly what we want since commit-map is mapped to *known-vis* by the state mapping, and we must show that the LUC postconditions describing how commit-maps are passed around hold when commit-maps are mapped to *known-vis*.

Recall that *known-vis* is defined to be the union of two *cmt-closure* operations (page 106), one over *conc-infer* and one over *seq-infer*. It would be convenient, in terms of the complexity of our proof, if we could prove monotonicity for each of these *cmt-closures* individually and thus conclude that their union also increases monotonically from operation to operation along a visible-affects chain. Unfortunately, it is not the case that each part increases monotonically. The descendant inferences that an earlier operation can make starting from its cmap may follow from a later operation's sequential inferences, and vice versa. The length and intricacy of the proof of Lemma 52 is a result of the way in which the sources of inferences switch from one operation to the next.

The proof of Lemma 52 consists in analyzing the sources of subsets of inferences for earlier operations and proving that the same inferences can be made for later operations by identifying their sources for the later operations. Thus we work with *cmt-closures* over subsets of *seq-infer* and *conc-infer*. Lemma 50 is an important technical lemma describing subset relationships between the kind of *cmt-closures* in which we interested. The definitions and lemmas that we now introduce, up through Lemma 48, will help to establish the fairly complicated hypotheses of Lemma 50.

A useful abstraction in the statement of Lemma 50 is that of a *return pair*. This is a pair of sets of transactions where the first set of the pair contains aborted transactions and the second set contains transactions that have returned (committed or aborted). For a pair of sets to comprise a return pair, the abort set must contain enough information to deduce which descendants of transactions in the return set have committed up through their ancestors in the return set. We used the idea of return pairs implicitly in our informal arguments of Section 3.5.2 when we argued that aborted sets contain enough information to allow us to make descendant inferences. In Section 4.5.1 the failure of particular combinations of committed and aborted sets to comprise return pairs lead us to conclude that some combinations resulted in incorrect protocols.

Definition 35. Let α be a well-formed IOC schedule, $\phi \in \alpha$, s a state of the IOC during α, and A and R sets of transactions. Then $\langle A, R \rangle$ is a *return pair for* ϕ *in* s if:

1. $\forall T \in (R - A).\text{COMMIT}(T, v)$ occurs before some instance of ϕ in α, and
2. $\forall T \in R.\forall T' \in desc(T).T' \in \text{aborted}(s) \Longrightarrow anc(T') \cap desc(T) \cap A \neq \emptyset$.

Lemma 36. Let α be a well-formed IOC schedule, $\phi \in \alpha$, s a state of the IOC during α, $\langle A, R\rangle$ a return pair for ϕ in s and $R' \subseteq R$. Then $\langle A, R'\rangle$ is a return pair for ϕ in s.

Proof: Easy from definitions. ■

The next lemma formally states the useful property of return pairs. Given a return pair $\langle A, R\rangle$, if a descendant of a transaction in R is ever aborted then it has an ancestor in A that is also a descendant of the transaction. In other words, A contains enough information to tell which descendants of transactions in R are aborted.[5]

Lemma 37. Let α be a well-formed IOC schedule, γ an IOC execution such that $schedule(\gamma) = \alpha$, ϕ an operation in α, s' a state immediately preceding the last instance of ϕ in γ or any time after that, $\langle A, R\rangle$ a return pair for ϕ in s', s the state immediately after γ, $A' \subseteq$ aborted(s). Then

$$\forall T \in desc(R).\ anc(T) \cap A' \cap desc(R) \neq \emptyset \Longrightarrow anc(T) \cap A \cap desc(R) \neq \emptyset.$$

Proof: See Appendix A. ■

The descendant inferences that we make include *cmt-closures* over *seq-infer* and *conc-infer*. In proving Lemma 52 we will need to apply Lemma 50 to show subset relationships between *cmt-closures* over subsets of *seq-infer* and *conc-infer*. For an operation π and state s we will need to establish that subsets of *seq-infer*(s, π) or *conc-infer*(s, π) combined with abort-map$(s)(\pi)$ comprise a return pair for π in s. This will show that abort-maps contain enough information to reach valid inferences starting either from the cmap or from sequential inferences.

As we mentioned, *seq-infer*(s, π) captures the reasoning about which transactions were sequentially before the transaction involved in π and therefore must have returned when π occurs. Now we want to prove that the abort-map for π in a state s and *seq-infer*(s, π) together comprise a return pair for π in s. This will tell us that if any descendant of a non-aborted transaction T in *seq-infer*(s, π) is ever going to have an aborted ancestor below T, then such an ancestor is in abort-map$(s)(\pi)$. We start by stating some smaller lemmas (proved in Appendix A) and then prove this property in Lemma 42.

The following lemma tells us that after a transaction T commits, each of its aborted descendants has an ancestor below T in the abort-map for the COMMIT operation.

Lemma 38. Let α be a well-formed IOC schedule, $\phi = \text{COMMIT}(T, v) \in \alpha$, and s a state of the IOC during α anytime after ϕ. Then

[5]The proof of this lemma makes use of Lemma 43, which is stated later.

$$\forall T' \in \textit{prop-desc}(T).T' \in \text{aborted}(s) \Longrightarrow \textit{anc}(T') \cap \text{abort-map}(s)(\phi) \cap \textit{prop-desc}(T) \neq \emptyset \ .$$

Proof: See Appendix A. ■

The next lemma says that if a transaction T is in the *seq-infer* set for an operation ϕ with $T_\phi = T'$, then the return for T prefix-affects ϕ.

Lemma 39. Let α be a well-formed IOC schedule, s a state of the IOC during α, ϕ any operation, and $T \in \textit{seq-infer}(s, \phi)$. Then a return operation for T prefix-affects all instances of ϕ in α.

Proof: See Appendix A. ■

The following lemma says that if an operation is prefix-affected by some return operation, then it knows about all aborts that the return operation knew about.

Lemma 40. Let α be an IOC schedule, γ an IOC execution such that $\textit{schedule}(\gamma) = \alpha$, ψ, ϕ in α such that ψ is a return operation that prefix-affects an instance of ϕ in α, s a state any time after the instance of ϕ prefix-affected by ψ in γ, and s' the state immediately after ψ in γ. Then $\text{abort-map}(s')(\psi) \subseteq \text{abort-map}(s)(\phi)$.

Proof: See Appendix A. ■

The next lemma is based upon the previous three. It says that if a transaction is in the *seq-infer* set for some operation, then all of the transaction's aborted descendants have an ancestor in the abort-map for the operation that is also a descendant of the transaction.

Lemma 41. Let α be a well-formed IOC schedule, ϕ any operation, s a state during α, $T \in \textit{seq-infer}(s, \phi)$. Then

$$\forall T' \in \textit{desc}(T).T' \in \text{aborted}(s) \Longrightarrow \textit{anc}(T') \cap \textit{desc}(T) \cap \text{abort-map}(s)(\phi) \neq \emptyset.$$

Proof: See Appendix A. ■

Now we can show that, for an operation ϕ in state s, $\text{abort-map}(s)(\phi)$ and $\textit{seq-infer}(s, \phi)$ comprise a return pair.

Lemma 42. Let α be a well-formed IOC schedule, s is a state of the IOC during α, $\phi \in \alpha$. Then $\langle \text{abort-map}(s)(\phi), \textit{seq-infer}(s, \phi) \rangle$ is a return pair for ϕ in s.

Proof: Note that the lemma is trivially true if s is a state before the first instance of ϕ in α. So assume s is a state any time after some instance of ϕ. We need to show two things:

1. $\forall T \in (\textit{seq-infer}(s,\phi) - \text{abort-map}(s)(\phi)).\text{COMMIT}(T,v)$ occurs before some instance of ϕ in α, and
2. $\forall T \in \textit{seq-infer}(s,\phi).\forall T' \in \textit{desc}(T).$
   $T' \in \text{aborted}(s) \Longrightarrow \textit{anc}(T') \cap \textit{desc}(T) \cap \text{abort-map}(s)(\phi) \neq \emptyset$.

Consider part 1. Let T be any transaction in $\textit{seq-infer}(s,\phi)$. By Lemma 39, a return operation for T prefix-affects every instance of ϕ in α. Call that operation ψ. Consider $\psi = \text{ABORT}(T)$. Let γ be an IOC execution containing s such that $\textit{schedule}(\gamma) = \alpha$. Let s' be the state immediately after ψ in γ. By Lemma 40, $\text{abort-map}(s')(\psi) \subseteq \text{abort-map}(s)(\phi)$. Thus it is easy to see from the IOC postconditions on ABORT that $T \in \text{abort-map}(s')(\psi)$, so $T \in \text{abort-map}(s)(\phi)$ also and there is nothing to show. If $\psi = \text{COMMIT}(T,v)$, then obviously the claim holds.

The proof of part 2 follows directly from Lemma 41. ■

We can prove a similar lemma about $\textit{conc-infer}(s,\pi)$ and $\text{abort-map}(s)(\pi)$ for an operation π in state s, namely, that these two sets also comprise a return pair for π in s. This tells us that if any descendant of a transaction T in $\textit{conc-infer}(s,\pi)$ has an aborted ancestor below T, then such an ancestor appears in $\text{abort-map}(s)(\pi)$. Again, we first state a few smaller lemmas (proved in Appendix A) and then the result in Lemma 45.

The following lemma says that when a transaction T commits, all of its descendants that have been created either have committed up to T or have some aborted ancestor below T.

Lemma 43. Let α be a well-formed IOC schedule, $\phi = \text{COMMIT}(T,v)$ an operation in α, and s' a state of the IOC immediately before ϕ. Then

$\forall T' \in \textit{prop-desc}(T) \cap \text{create-requested}(s').$
either $\textit{anc}(T') \cap \textit{prop-desc}(T) \cap \text{aborted}(s') \neq \emptyset$, or
$\textit{anc}(T') \cap \textit{prop-desc}(T) \subseteq \text{committed}(s')$.

Proof: See Appendix A. ■

The next lemma is analogous to Lemma 41 and is based on the previous lemma. It says that if a transaction T is in the *conc-infer* set for some operation, then all of T's aborted descendants have an ancestor that is also a descendant of T in the abort-map for the operation.

Lemma 44. Let α be a well-formed IOC schedule, ϕ any operation, s a state during α, $T \in \textit{conc-infer}(s,\phi)$. Then

$\forall T' \in \textit{desc}(T).T' \in \text{aborted}(s) \Longrightarrow \textit{anc}(T') \cap \textit{desc}(T) \cap \text{abort-map}(s)(\phi) \neq \emptyset.$

Proof: See Appendix A. ■

Now we can prove that abort-map$(s)(\pi)$ and *conc-infer*(s, π) comprise a return pair for π in s.

Lemma 45. Let α be a well-formed IOC schedule, s is a state of the IOC during α, $\phi \in \alpha$. Then $\langle$abort-map$(s)(\phi)$, *conc-infer*$(s, \phi)\rangle$ is a return pair for ϕ in s.

Proof: Note that the lemma is trivially true if s is a state before the first instance of ϕ in α. So assume s is a state any time after some instance of ϕ. We need to show two things:

1. $\forall T \in (\textit{conc-infer}(s, \phi) - \text{abort-map}(s)(\phi)).\text{COMMIT}(T, v)$ occurs before some instance of ϕ in α, and
2. $\forall T \in \textit{conc-infer}(s, \phi).\forall T' \in \textit{desc}(T).$
   $T' \in \text{aborted}(s) \Longrightarrow \textit{anc}T' \cap \textit{desc}(T) \cap \text{abort-map}(s)(\phi) \neq \emptyset$.

For part 1, consider any $T \in \textit{conc-infer}(s, \phi) - \text{abort-map}(s)(\phi)$. Let γ be an IOC execution containing s such that $\textit{schedule}(\gamma) = \alpha$. Let ϕ' be the last instance of ϕ in γ before s, and let s' be the state immediately after ϕ' in γ. By Lemma 29, $\text{cmap}(s')(\phi) = \text{cmap}(s)(\phi)$. Thus $\textit{conc-infer}(s, \phi) = \textit{conc-infer}(s', \phi)$ and $T \in \textit{conc-infer}(s', \phi)$. By Lemma 27, $\textit{conc-infer}(s', \phi) \subseteq \text{committed}(s')$. Thus, by Lemma 28, $\text{COMMIT}(T, v)$ occurs before s' and hence before ϕ'.

Part 2 follows directly from Lemma 44. ■

The following definitions are used in the statement of Lemma 50. They describe relationships between sets of transactions.

Definition 46. Let $\mathcal{S}$ and $\mathcal{T}$ be either sets of transactions or individual transactions. Then define

$$\textit{anc-between}(\mathcal{S}, \mathcal{T}) = \textit{anc}(\mathcal{S}) \cap (\textit{desc}(\mathcal{T}) - \textit{desc}(\mathcal{S})).$$

Definition 47. Let $\mathcal{S}$ and $\mathcal{T}$ be sets of transactions. We say that $\mathcal{T}$ *dominates* $\mathcal{S}$ if $\forall T \in \mathcal{S}.\, \textit{anc}(T) \cap \mathcal{T} \neq \emptyset$.

Lemma 48. If $\mathcal{S} \subseteq \mathcal{T}$ then $\mathcal{T}$ dominates $\mathcal{S}$.

Proof: Easy from definitions. ■

The next lemma states a general property of *cmt-closure*. It is used prove Lemma 50.

Lemma 49. Let C_1, C_2, A_1, A_2, R_1, R_2 be sets of transactions such that the following conditions hold:

C1. $C_1 \subseteq C_2$, and

C2. R_2 dominates R_1, and

C3. $anc\text{-}between(R_1, R_2) \cap A_2 = \emptyset$, and

C4. $\forall T \in desc(R_1).\ anc(T) \cap A_2 \cap desc(R_1) \neq \emptyset \Longrightarrow anc(T) \cap A_1 \cap desc(R_1) \neq \emptyset$.

Then $cmt\text{-}closure(C_1, A_1, R_1) \subseteq cmt\text{-}closure(C_2, A_2, R_2)$.

Proof: See Appendix A. ■

Now we can state and prove the technical lemma about *cmt-closures* that will be used repeatedly in the proof of Lemma 52. In all cases when we apply the lemma, $\mathcal{S}$ will be a subset of either $seq\text{-}infer(s', \phi)$ or $conc\text{-}infer(s', \phi)$, and $\mathcal{T}$ will be a subset of either $seq\text{-}infer(s, \pi)$ or $conc\text{-}infer(s, \pi)$. We state the hypotheses in terms of return-pairs, *etc.*, for the sake of generality and to isolate the properties of the sets $\mathcal{S}$ and $\mathcal{T}$ relied upon by the lemma.

Lemma 50. Let $\alpha\pi$ be a well-formed IOC schedule, $\gamma = \gamma'\pi s$ an IOC execution such that $schedule(\gamma) = \alpha\pi$, ϕ an operation in α, s' the state at the end of γ', $\mathcal{S}$ and $\mathcal{T}$ sets of transactions, and assume that the following conditions hold:

C1. $\mathcal{T}$ dominates $\mathcal{S}$, and

C2. $anc\text{-}between(\mathcal{S}, \mathcal{T}) \cap \text{abort-map}(s)(\pi) = \emptyset$, and

C3. $\langle \text{abort-map}(s')(\phi), \mathcal{S} \rangle$ is a return pair for ϕ in s' .

Then

$$cmt\text{-}closure(\text{created}(s'), \text{abort-map}(s')(\phi), \mathcal{S}) \subseteq cmt\text{-}closure(\text{created}(s), \text{abort-map}(s)(\pi), \mathcal{T}) .$$

Proof: The result follows from Lemma 49. The conditions for applying the lemma are satisfied as follows:

C1. $\text{created}(s') \subseteq \text{created}(s)$ by Lemma 31, and

C2. $\mathcal{T}$ dominates $\mathcal{S}$ by this lemma's own C1, and

C3. $anc\text{-}between(\mathcal{S}, \mathcal{T}) \cap \text{abort-map}(s)(\pi) = \emptyset$ by this lemma's own C2, and

C4. Follows from Lemma 37, which applies by own C3, and IOC postcondition which specifies $\text{abort-map}(s)(\pi) \subseteq \text{aborted}(s)$.

■

We state one more definition before finally arriving at Lemma 52. The definition describes a property of a set of transactions.

Definition 51. A set of transactions $\mathcal{T}$ is *mutually unrelated* if $\forall T \in \mathcal{T}.\ anc(T) \cap \mathcal{T} = \{T\}$. That is, T has no ancestor in $\mathcal{T}$ other than itself. Clearly, any subset of a mutually unrelated set is itself mutually unrelated.

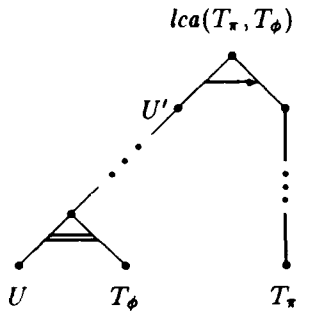


Figure 6.1: Example of non-monotonicity in sources of inferences.

We will now prove Lemma 52, which states that if an operation ϕ directly-visible-affects an operation π then π can infer at least the same commits that ϕ could. This lemma is a key part of the proof that the IOC simulates the LUC. The proof of the lemma consists of a lengthy case analysis. In the interests of readability, many of the arguments for the cases have been extracted into lemmas. These lemmas are stated and proved (where necessary) in Appendix A.

Lemma 52. Let $\alpha\pi$ be a well-formed IOC schedule, ϕ an operation such that an instance of ϕ directly-visible-affects an instance of π in $\alpha\pi$, s' a state of the IOC immediately after α, s a state of the IOC immediately after $\alpha\pi$, (s', π, s) a step of the IOC. Then $\textit{known-vis}(s', \phi) \subseteq \textit{known-vis}(s, \pi)$.

Proof: By case analysis. Let C_1 = created(s'), C_2 = created(s), A_1 = abort-map(s')(ϕ), and A_2 = abort-map(s)(π). Expanding the definition of *known-vis*, we want to show that for any ϕ, π such that ϕ directly-visible-affects π in $\alpha\pi$:

$$\begin{aligned}&\textit{cmt-closure}(C_1, A_1, \textit{seq-infer}(s', \phi)) \cup \textit{cmt-closure}(C_1, A_1, \textit{conc-infer}(s', \phi))\\ &\quad \subseteq \textit{cmt-closure}(C_2, A_2, \textit{seq-infer}(s, \pi)) \cup \textit{cmt-closure}(C_2, A_2, \textit{conc-infer}(s, \pi))\ .\end{aligned}$$

For some cases of operations π and ϕ the inferences that can be made starting from *seq-infer* and *conc-infer* are each monotonic from state s' to state s and we can prove this directly. In other cases, transactions that were in the *cmt-closure* starting from $\textit{seq-infer}(s', \phi)$ will be in the *cmt-closure* starting from $\textit{conc-infer}(s, \pi)$ or those that were in the *cmt-closure* starting from $\textit{conc-infer}(s', \phi)$ will be in *cmt-closure* starting from $\textit{seq-infer}(s, \pi)$. These other cases require a detailed analysis to show how transactions move between sequential inference sets and concurrent inference sets.

For examples of how sources of inferences switch from *seq-infer* to *conc-infer* and vice versa consider Figure 6.1, which depicts a possible transaction tree relating T_π and T_ϕ. Assume ϕ = REQUEST-COMMIT(T_ϕ, v') and π = REQUEST-COMMIT(T_π, v), where T_π and T_ϕ are accesses to an object X and T_ϕ is locally visible to T_π at X. In the figure, U is in $\textit{conc-infer}(s', \phi)$ (and thus is in $\textit{cmt-closure}(C_1, A_1, \textit{conc-infer}(s', \phi))$)

but not in $conc\text{-}infer(s,\pi)$. However, U', an ancestor of U, is in $seq\text{-}infer(s,\pi)$. Since U is committed up to $lca(U,T_\phi)$ and T_ϕ is committed up to U', we know that U is in $cmt\text{-}closure(C_2, A_2, seq\text{-}infer(s,\pi))$. So the source of inferences switches from *conc-infer* to *seq-infer*. If we change Figure 6.1 so that U is sequential with T_ϕ and U' is concurrent with the ancestor of T that is a child of $lca(T_\pi, T_\phi)$ then we have the opposite situation. In that case U is in $seq\text{-}infer(s',\phi)$ but not in $seq\text{-}infer(s,\pi)$. However, U' will be in $conc\text{-}infer(s,\pi)$ and thus if U is in $cmt\text{-}closure(C_1, A_1, seq\text{-}infer(s',\phi))$ then U will be in $cmt\text{-}closure(C_2, A_2, conc\text{-}infer(s,\pi))$.

Proceeding with the proof, let $T'' = lca(T_\phi, T_\pi)$. It is easy to see from the definitions that we can split *conc-infer* (or *seq-infer*) into two parts around T'', as follows:

$$conc\text{-}infer(s',\phi) = cpa(T'', conc\text{-}infer(s',\phi)) \cup cd(T'', conc\text{-}infer(s',\phi)), \text{ and}$$

$$seq\text{-}infer(s',\phi) = ca(T'', seq\text{-}infer(s',\phi)) \cup cpd(T'', seq\text{-}infer(s',\phi)).$$

By Lemma 61, the *cmt-closure* over the LHS of each equation above is equal to the union of the separate *cmt-closures* over each of the RHS sets. Thus we can consider the *cmt-closures* over each of these subsets separately.

The proof proceeds as follows. For each case considered, we find pairs $\langle \mathcal{S}_i, \mathcal{T}_i \rangle$ such that:

$$cmt\text{-}closure(C_1, A_1, \mathcal{S}_i) \subseteq cmt\text{-}closure(C_2, A_2, \mathcal{T}_i)$$

and

$$\bigcup_i \mathcal{S}_i = seq\text{-}infer(s',\phi) \cup conc\text{-}infer(s',\phi), \text{ and}$$
$$\bigcup_i \mathcal{T}_i = seq\text{-}infer(s,\pi) \cup conc\text{-}infer(s,\pi)\ .$$

The assignments to $\mathcal{S}$ and $\mathcal{T}$ are:

For π =REQUEST-COMMIT(T,v), T an access to X, ϕ locally visible to π at X in α, $\phi \neq$ CREATE(T), or
π = REQUEST-COMMIT(T,v), $location(\phi) = T$, $\phi \neq$ CREATE(T), or
π = REQUEST-CREATE(T), $location(\phi) = parent(T)$, $\phi \neq$ CREATE$(parent(T))$.
If T and T_ϕ are descendants of sequential siblings or $T = parent(T_\phi)$, then
a. $\mathcal{S} = seq\text{-}infer(s',\phi)$, $\mathcal{T} = seq\text{-}infer(s,\pi)$
b. $\mathcal{S} = cpa(T'', conc\text{-}infer(s',\phi))$, $\mathcal{T} = cpa(T'', conc\text{-}infer(s,\pi))$
c. $\mathcal{S} = cd(T'', conc\text{-}infer(s',\phi))$, $\mathcal{T} = c(T'', seq\text{-}infer(s,\pi))$
If T and T_ϕ are descendants of concurrent siblings, then
a. $\mathcal{S} = conc\text{-}infer(s',\phi)$, $\mathcal{T} = conc\text{-}infer(s,\pi)$
b. $\mathcal{S} = ca(T'', seq\text{-}infer(s',\phi))$, $\mathcal{T} = ca(T'', seq\text{-}infer(s,\pi))$
c. $\mathcal{S} = cpd(T'', seq\text{-}infer(s',\phi))$, $\mathcal{T} = c(T'', conc\text{-}infer(s,\pi))$
For π = REQUEST-CREATE(T), ϕ = CREATE(T''), $T'' = parent(T)$, or
π = REQUEST-COMMIT(T,v), ϕ = CREATE(T), or
π = CREATE(T), ϕ = REQUEST-CREATE(T), or

π = COMMIT(T, v), ϕ = REQUEST-COMMIT(T, v), or
π = ABORT(T), ϕ = REQUEST-CREATE(T), or
π = INFORM-COMMIT-AT(X)OF(T), ϕ = COMMIT(T, v), or
π = INFORM-ABORT-AT(X)OF(T), ϕ = ABORT(T).
a. $\mathcal{S} = \textit{seq-infer}(s', \phi)$, $\mathcal{T} = \textit{seq-infer}(s, \pi)$
b. $\mathcal{S} = \textit{conc-infer}(s', \phi)$, $\mathcal{T} = \textit{conc-infer}(s, \pi)$

The proof in each case above will follow directly from Lemma 50. We must simply establish that the conditions for applying the lemma hold. We will establish condition C3 once at the beginning and then consider conditions C1 and C2 case by case.

C3 requires that $\langle \text{abort-map}(s')(\phi), \mathcal{S} \rangle$ be a return pair for ϕ in s. Note that in all cases above, $\mathcal{S}$ and $\mathcal{T}$ are subsets of either $\textit{seq-infer}(s', \phi)$ or $\textit{conc-infer}(s', \phi)$. If they are subsets of $\textit{seq-infer}(s', \phi)$ Lemmas 36 and 42 establish the condition. If they are subsets of $\textit{conc-infer}(s', \phi)$ Lemmas 36 and 45 establish it.

Now we will consider cases for π to establish:

C1: $\mathcal{T}$ dominates $\mathcal{S}$, and

C2: $\textit{anc-between}(\mathcal{S}, \mathcal{T}) \cap \text{abort-map}(s)(\pi) = \emptyset$.

For C1, we will prove that either $\mathcal{S} \subseteq \mathcal{T}$, or $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2$, $\mathcal{S}_1 \subseteq \mathcal{T}$, and $\mathcal{S}_2$ is dominated by $\{T\}$ for some $T \in \mathcal{T}$. The result then follows from Lemma 48 and the definition of dominate. For C2, if $\mathcal{S} \subseteq \mathcal{T}$ and hence $\textit{anc-between}(\mathcal{S}, \mathcal{T}) = \emptyset$, then clearly $\textit{anc-between}(\mathcal{S}, \mathcal{T}) \cap \text{abort-map}(s)(\pi) = \emptyset$. Otherwise, if $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2$, then Lemma 69 allows us to consider each subset separately. For $\mathcal{S}_1$, $\textit{anc-between}(\mathcal{S}_1, \mathcal{T}) = \emptyset$, as above. For $\mathcal{S}_2$, which is dominated by $\{T\}$, we must show that no transaction in $\textit{anc-between}(\mathcal{S}_2, \{T\})$ is in abort-map$(s)(\pi)$.

1. π =REQUEST-COMMIT(T, v), T an access to X, ϕ locally visible to π at X in α, $\phi \neq$ CREATE(T), or
2. π = REQUEST-COMMIT(T, v), $\textit{location}(\phi) = T$, $\phi \neq$ CREATE(T), or
3. π = REQUEST-CREATE(T), $\textit{location}(\phi) = \textit{parent}(T)$, $\phi \neq$ CREATE$(\textit{parent}(T))$.

If T and T_ϕ are descendants of sequential siblings or $T = \textit{parent}(T_\phi)$:

a. $\mathcal{S} = \textit{seq-infer}(s', \phi)$, $\mathcal{T} = \textit{seq-infer}(s, \pi)$
Let U' be the ancestor of T_ϕ that is a child of T''.

C1: By Lemma 80, $\mathcal{S}$ is a set of children of ancestors of T_ϕ. So by Lemma 72, $\mathcal{S} = ca(T'', \mathcal{S}) \cup cd(U', \mathcal{S})$. By Lemma 74, $ca(T'', \mathcal{S}) \subseteq ca(T'', \mathcal{T})$, so certainly, $ca(T'', \mathcal{S}) \subseteq \mathcal{T}$. Since T and T_ϕ are descendants of sequential siblings and ϕ occurs before π, U' is a prior sequential sibling of an ancestor of T and hence is in $\mathcal{T}$. So any $T' \in cd(U', \mathcal{S})$ has an ancestor in $\mathcal{T}$, namely, U'.

C2: By Lemma 65, $\mathcal{T}$ is mutually unrelated, so $anc\text{-}between(ca(T'', \mathcal{S}), \mathcal{T}) = \emptyset$ by Lemma 67. In case 1, $anc\text{-}between(cd(U', \mathcal{S}), \{U'\})$ are all ancestors of T_ϕ that are descendants of U', by Lemma 68 and these cannot be in abort-map$(s)(\pi)$ by Lemmas 70 and 71. For cases 2 and 3, $T_\phi = U'$ so $cd(U', \mathcal{S})$ contains only children of T_ϕ. If $cd(U', \mathcal{S})$ is empty then it easy to see from the definitions that $anc\text{-}between(cd(U', \mathcal{S}), \mathcal{T}) = \emptyset$. Otherwise, $anc\text{-}between(cd(U', \mathcal{S}), \{U'\}) = T_\phi$ and T_ϕ cannot be in abort-map$(s)(\pi)$ by Lemmas 75 and 71.

b. $\mathcal{S} = cpa(T'', conc\text{-}infer(s', \phi))$, $\mathcal{T} = cpa(T'', conc\text{-}infer(s, \pi))$

C1: By controller postconditions on π, $conc\text{-}with(T, cpa(T, \text{cmap}(s')(\phi))) \subseteq \text{cmap}(s)(\pi)$. So by Lemma 76, $\mathcal{S} \subseteq \mathcal{T}$.

C2: By Lemma 66, $\mathcal{T}$ is mutually unrelated, so by Lemma 67, $anc\text{-}between(\mathcal{S}, \mathcal{T}) = \emptyset$.

c. $\mathcal{S} = cd(T'', conc\text{-}infer(s', \phi))$, $\mathcal{T} = c(T'', seq\text{-}infer(s, \pi))$
Let U' be the ancestor of T_ϕ that is a child of T''.

C1: Obviously from the definition, $conc\text{-}infer(s', \phi)$ is a set of children of proper ancestors of T_ϕ. So by Lemma 77,

$$\mathcal{S} = c(T'', conc\text{-}infer(s', \phi)) \cup cd(U', conc\text{-}infer(s', \phi)).$$

By Lemma 79, $c(T'', conc\text{-}infer(s', \phi)) \subseteq \mathcal{T}$. In cases 2 and 3, by Corollary 78, $\mathcal{S} = c(T'', conc\text{-}infer(s', \phi))$, so we can ignore $cd(U', conc\text{-}infer(s', \phi))$. In case 1, $U' \in seq\text{-}infer(s, \pi)$ since it is a prior sequential sibling of an ancestor of T, so every transaction in $cd(U', conc\text{-}infer(s', \phi))$ has an ancestor in $\mathcal{T}$, namely, U'.

C2: By Lemma 65, $\mathcal{T}$ is mutually unrelated. So by Lemma 67, $anc\text{-}between(c(T'', conc\text{-}infer(s', \phi)), \mathcal{T}) = \emptyset$. For case 1, from the definitions we can see that all transactions in $cd(U', conc\text{-}infer(s', \phi))$ are children of ancestors of T_ϕ that are descendants of U'. So $anc\text{-}between(cd(U', conc\text{-}infer(s', \phi)), \{U'\})$ are all ancestors of T_ϕ below T'' by Lemma 68. They cannot be in abort-map$(s)(\pi)$ by Lemmas 70 and 71.

If T and T_ϕ are descendants of concurrent siblings (only possible in cases 1 and 3 for π):

a. $\mathcal{S} = conc\text{-}infer(s', \phi)$, $\mathcal{T} = conc\text{-}infer(s, \pi)$
Let U' be the ancestor of T_ϕ that is a child of T''. From the definitions, $\mathcal{S}$ is a set of children of ancestors of T_ϕ. So by Lemma 72,

$$\mathcal{S} = ca(T'', \mathcal{S}) \cup cd(U', \mathcal{S}).$$

C1: By the IOC postconditions on π, $conc\text{-}with(T, cpa(T, \text{cmap}(s')(\phi))) \subseteq \text{cmap}(s)(\pi)$. By Lemma 83, $ca(T'', \mathcal{S}) \subseteq \mathcal{T}$. For case 3, $\mathcal{S} = ca(T'', \mathcal{S})$

by Corollary 73 so we need not consider $cd(U', \mathcal{S})$. For case 1, by Lemma 84, $U' \in \mathcal{T}$. Thus every transaction in $cd(U', \mathcal{S})$ has an ancestor in $\mathcal{T}$, namely, U'.

C2: $\mathcal{T}$ is mutually unrelated by Lemma 66, so $anc\text{-}between(ca(T'', \mathcal{S}), \mathcal{T}) = \emptyset$, by Lemma 67. For case 1, $anc\text{-}between(cd(U', \mathcal{S}), \{U'\})$ are all ancestors of T_ϕ that are descendants of U' by Lemma 68 and hence cannot be in abort-map$(s)(\pi)$ by Lemmas 70 and 71.

b. $\mathcal{S} = ca(T'', seq\text{-}infer(s', \phi))$, $\mathcal{T} = ca(T'', seq\text{-}infer(s, \pi))$

C1: By Lemma 74, $\mathcal{S} \subseteq \mathcal{T}$.

C2: By Lemma 65, $\mathcal{T}$ is mutually unrelated. So By Lemma 67, $anc\text{-}between(\mathcal{S}, \mathcal{T}) = \emptyset$.

c. $\mathcal{S} = cpd(T'', seq\text{-}infer(s', \phi))$, $\mathcal{T} = c(T'', conc\text{-}infer(s, \pi))$
Let U' be the ancestor of T_ϕ that is a child of T''. By Lemma 80, $seq\text{-}infer(s', \phi)$ is a set of children of ancestors of T_ϕ, so by Lemma 81, $\mathcal{S} = cd(U', seq\text{-}infer(s', \phi))$. For case 3, $U' = T_\phi$, so by Corollary 82, $\mathcal{S} = c(T_\phi, seq\text{-}infer(s', \phi))$.

C1: For case 1, by Lemma 84, $U' \in conc\text{-}infer(s, \pi)$ and so $U' \in \mathcal{T}$. For case 3, if $\mathcal{S}$ is empty then trivially, $\mathcal{T}$ dominates $\mathcal{S}$. Otherwise, by Lemma 75, ϕ is a COMMIT operation and by Lemma 85, $T_\phi \in \mathcal{T}$. Thus in both cases every transaction in $\mathcal{S}$ has an ancestor in $\mathcal{T}$, namely, U'.

C2: For case 1, by Lemma 80 and the definition of *cpd*, $\mathcal{S}$ contains children of ancestors of T_ϕ that are descendants of U'. So $anc\text{-}between(\mathcal{S}, \{U'\})$ are all ancestors of T_ϕ below T'' by Lemma 68 and hence cannot be in abort-map$(s)(\pi)$ by Lemmas 70 and 71. For case 3, if $\mathcal{S}$ is empty then it is easy to see from the definition that $anc\text{-}between(\mathcal{S}, T_\phi) = \emptyset$. Otherwise $anc\text{-}between(\mathcal{S}, \{T_\phi\}) = T_\phi$, and T_ϕ cannot be in abort-map$(s)(\pi)$ by Lemmas 75 and 71.

4. π = REQUEST-CREATE(T), ϕ = CREATE(T''), $T'' = parent(T)$, or
5. π = REQUEST-COMMIT(T, v), ϕ = CREATE(T), or
6. π = CREATE(T), ϕ = REQUEST-CREATE(T), or
7. π = COMMIT(T, v), ϕ = REQUEST-COMMIT(T, v), or
8. π = ABORT(T), ϕ = REQUEST-CREATE(T), or
9. π = INFORM-COMMIT-AT(X)OF(T), ϕ = COMMIT(T, v), or
10. π = INFORM-ABORT-AT(X)OF(T), ϕ = ABORT(T).

a. $\mathcal{S} = conc\text{-}infer(s', \phi)$, $\mathcal{T} = conc\text{-}infer(s, \pi)$

C1: For cases 4 and 5, $conc\text{-}with(T, cpa(T, \text{cmap}(s')(\phi))) \subseteq \text{cmap}(s)(\pi)$ by the controller postconditions. For cases 6 through 10, by the controller postconditions, $\text{cmap}(s')(\phi) \subseteq \text{cmap}(s)(\pi)$. In all cases, from the definition of *conc-infer*, it is easy to see that $\mathcal{S} \subseteq \mathcal{T}$ (the only case that is not trivial is 4, which requires a simple inference from the definition).

C2: $\mathcal{T}$ is mutually unrelated by Lemma 66, so $anc\text{-}between(\mathcal{S}, \mathcal{T}) = \emptyset$ by Lemma 67.

b. $\mathcal{S} = seq\text{-}infer(s', \phi)$, $\mathcal{T} = seq\text{-}infer(s, \pi)$

C1: For the cases where $T_\phi = T_\pi$ it is easy to see from the definition of $seq\text{-}infer()$ that $\mathcal{S} \subseteq \mathcal{T}$. In case 4, it is easy to see that any sibling of an ancestor of T'' is also a sibling of an ancestor of T, so $\mathcal{S} \subseteq \mathcal{T}$.

C2: $\mathcal{T}$ is mutually unrelated by Lemma 65, so $anc\text{-}between(\mathcal{S}, \mathcal{T}) = \emptyset$ by Lemma 67.

■

We need just a few more lemmas before we can prove the simulation theorem for this section. The next lemma says, in effect, that an operation cannot "lose" inferences. Once a transaction is in the sequential or concurrent inferences for an operation, it will always be there. The lemma is used in the proof of the simulation theorem to show that the first LUC postcondition on all operations holds under the state mapping.

Lemma 53. Let $\alpha\pi$ be a well-formed IOC schedule, s' a state of the IOC immediately after α, s a state of the IOC immediately after $\alpha\pi$, π a single operation, and (s', π, s) a step of the IOC. Then

$$\forall \phi.\ all\text{-}seq(s', \phi) \subseteq all\text{-}seq(s, \phi), \text{ and}$$

$$\forall \phi.\ all\text{-}conc(s', \phi) \subseteq all\text{-}conc(s, \phi).$$

Proof: Note that if $\phi \notin \alpha$ then the results are immediate. Otherwise, the results will follow from Lemma 50 once the conditions for applying the lemma are established. Let ϕ be an operation in α. Let $C_1 = \text{created}(s')$, $C_2 = \text{created}(s)$, $A_1 = \text{abort-map}(s')(\phi)$, $A_2 = \text{abort-map}(s)(\phi)$.

First, expanding the definition of *all-seq*, we want to show

$$cmt\text{-}closure(C_1, A_1, seq\text{-}infer(s', \phi)) \subseteq cmt\text{-}closure(C_2, A_2, seq\text{-}infer(s, \phi)).$$

Condition C3 holds by Lemma 42. For C1, it is easy to see from the definition of *seq-infer* that $seq\text{-}infer(s', \phi) \subseteq seq\text{-}infer(s, \phi)$. C2 then follows from Lemma 67.

Second, expanding the definition of *all-conc*, we want to show

$$cmt\text{-}closure(C_1, A_1, conc\text{-}infer(s', \phi)) \subseteq cmt\text{-}closure(C_2, A_2, conc\text{-}infer(s, \phi)).$$

Condition C3 holds by Lemma 45. For C1, it is easy to see that $conc\text{-}infer(s', \phi) \subseteq conc\text{-}infer(s, \phi)$ since the controller postconditions specify that $\text{cmap}(s')(\phi) \subseteq \text{cmap}(s)(\phi)$ if $\phi = \pi$ and $\text{cmap}(s')(\phi) = \text{cmap}(s)(\phi)$ otherwise. Again, C2 follows from Lemma 67. ■

The next lemma and the following two corollaries prove that the *cmt-closures* in which we are interested actually contain only committed transactions. They use the properties of return pairs discussed earlier. The corollaries are used in the proof of the simulation theorem to show that the fourth LUC postcondition on all operations holds under the state mapping.

Lemma 54. Let α be a well-formed IOC schedule, s a state of the IOC immediately after α, ϕ any operation in α, $\langle A, R\rangle$ a return pair for ϕ in s, $C \subseteq \text{created}(s)$. Then $\textit{cmt-closure}(C, A, R) \subseteq \text{committed}(s)$.

Proof: First, note that if $R = \emptyset$ then the result holds trivially. So assume $R \neq \emptyset$. Since $\langle A, R\rangle$ is a return pair, we know:

F1. $\forall T \in (R - A).\text{COMMIT}(T, v)$ occurs before some instance of ϕ in α, and

F2. $\forall T \in R.\forall T' \in \textit{desc}(T).T' \in \text{aborted}(s) \Longrightarrow \textit{anc}(T') \cap \textit{desc}(T) \cap A \neq \emptyset$.

Expanding the definition of *cmt-closure*, we must show:

$$C \cap \textit{desc}(R) \cap \textit{non-orphans}(A, R) \subseteq \text{committed}(s).$$

Consider any $T \in C \cap \textit{desc}(R) \cap \textit{non-orphans}(A, R)$. Let T' be an ancestor of T in R. By definition of *non-orphans* $\textit{anc}(T) \cap \textit{desc}(R) \cap A = \emptyset$. Thus $T' \notin A$. By F1, we know that $\psi = \text{COMMIT}(T', v)$ occurs before some instance of ϕ in α. Now, if $T' = T$ then, by Lemma 25, $T \in \text{committed}(s)$ and we are done. So assume $T' \neq T$. By the contrapositive of F2, we know $\forall T'' \in \textit{desc}(T').\ \textit{anc}(T'') \cap \textit{desc}(T') \cap A = \emptyset \Longrightarrow T'' \notin \text{aborted}(s)$. Thus no descendant of T' that is an ancestor of T is in aborted(s). Let γ be an IOC execution containing s such that $\textit{schedule}(\gamma) = \alpha$. Let s' be the state just before ψ in γ. By Lemma 43,

$$\begin{aligned}&\forall T'' \in \textit{prop-desc}(T') \cap \text{create-requested}(s').\\ &\quad \text{either } \textit{anc}(T'') \cap \textit{prop-desc}(T') \cap \text{aborted}(s') \neq \emptyset\\ &\quad \text{or } \quad \textit{anc}(T'') \cap \textit{prop-desc}(T') \subseteq \text{committed}(s') \ .\end{aligned}$$

In particular, this holds for $T'' = T$. But if no descendant of T' that is an ancestor of T is in aborted(s), then certainly no proper descendant of T' that is an ancestor of T is in aborted(s') (by Lemma 31). Thus it must be that all proper descendants of T' that are ancestors of T are in committed(s') and hence in committed(s) (by Lemma 31). So, $T \in \text{committed}(s)$, as required. ■

Corollary 55. Let α be a well-formed IOC schedule, s a state of the IOC immediately after α, ϕ any operation. Then $\textit{all-seq}(s, \phi) \subseteq \text{committed}(s)$.

Proof: If $\phi \notin \alpha$ then the result holds trivially. Otherwise the result follows directly from Lemmas 54 and 42. ■

Corollary 56. Let α be a well-formed IOC schedule, s a state of the IOC immediately after α, ϕ any operation. Then $\mathit{all\text{-}conc}(s, \phi) \subseteq \text{committed}(s)$.

Proof: If $\phi \notin \alpha$ then the result holds trivially. Otherwise the result follows directly from Lemmas 54 and 45. ■

Now we can prove the simulation theorem for this section, that every well-formed schedule of the IOC is also a schedule of the LUC. Essentially, we show that the IOC can be considered as a particular implementation of the LUC, where the commit-map for an operation in a state of the LUC contains exactly those transactions that the operation can infer to have committed in the corresponding state of the IOC.

Theorem 57. Every well-formed IOC schedule is an LUC schedule.

Proof: Let α be a well-formed IOC schedule and let s be a state of the IOC immediately after α. Let t be a state of the LUC that satisfies the following state mapping:

$\forall \phi.\, \text{commit-map}(t)(\phi) = \mathit{known\text{-}vis}(s, \phi)$, and
every other component of t is equal to the corresponding component of s.

We want to show that α is a schedule of the LUC and that t is a state of the LUC immediately after α.

The proof is by induction on the length of α. The basis, α of length 0, holds trivially (it is easy to show that the initial state of the LUC satisfies the state mapping). For the induction hypothesis, assume that the claim holds for α of length $n-1$. We'll show that the claim holds for $\alpha = \alpha'\pi$, a well-formed IOC schedule of length n, where π is a single operation.

Let s' be a state of the IOC immediately after α' in which π is enabled. By the induction hypothesis, α' is also a schedule of the LUC. So let t' be a state of the LUC after α' that satisfies the state mapping (such a state exists by the induction hypothesis). We must show that (t', π, t) is a step of the LUC. By Lemma 34, since π is enabled in the IOC in state s', π is enabled in the LUC in state t'. Now it just remains to show that the postconditions for the LUC after π are satisfied in state t.

First, it should be easy to see that all postconditions involving state components other that commit-map are satisfied in state t since they are satisfied in state s and the components of state t are the same as the corresponding components of state s under the state mapping.

The LUC postconditions on all operations involving commit-map require

$$\forall \phi.\, \text{commit-map}(t')(\phi) \subseteq \text{commit-map}(t)(\phi), \text{ and}$$

$$\forall \phi.\, \text{commit-map}(t)(\phi) \subseteq \text{committed}(t).$$

Thus, using the state mapping, we need to show

$$\forall \phi.\, \textit{known-vis}(s', \phi) \subseteq \textit{known-vis}(s, \phi), \text{ and}$$

$$\forall \phi.\, \textit{known-vis}(s, \phi) \subseteq \text{committed}(s).$$

The first follows directly from Lemma 53 and the second from Corollaries 55 and 56 and the definition of *known-vis*.

All cases of the postconditions on π involving commit-map will follow from Lemma 52, since the postconditions of the LUC involving commit-map all are equivalent to

$$\forall \phi.\text{an instance of } \phi \text{ directly-visible-affects } \pi \text{ in } \alpha\pi \Longrightarrow \text{commit-map}(t')(\phi) \subseteq \text{commit-map}(t)(\pi),$$

which, using the state mapping, translates to

$$\forall \phi.\text{an instance of } \phi \text{ directly-visible-affects } \pi \text{ in } \alpha\pi \Longrightarrow \textit{known-vis}(s', \phi) \subseteq \textit{known-vis}(s, \pi)\ .$$

■

6.3.5 *Correctness of IO Systems*

Two simple corollaries of our two simulation theorems are that the IOC simulates the GKC, and that IO Systems strongly simulate GK Systems.

Corollary 58. Every well-formed schedule of the IOC is a schedule of the GKC.

Proof: Immediate from Theorems 57 and 24. ■

Corollary 59. IO Systems strongly simulate GK Systems in that every IO System schedule is a GK System schedule.

Finally, we can prove that IO Systems guarantee eager diffusion. The result follows immediately from the preceding corollary and the results in Section 6.1.3.

Theorem 60. IO Systems guarantee eager diffusion.

Proof: Immediate from Theorem 18, Corollary 59, and Theorem 19. ■

6.4 Discussion of Inference Optimized Systems

To summarize, we defined Inference Optimized Systems to model the sequential and descendant inferences used in the simplified version of our protocol presented in Chapter 3. We then proved that Inference Optimized Systems simulate Global Knowledge

Systems in the strong sense that every schedule of the former system is a schedule of the latter. This allowed us to conclude that Inference Optimized Systems guarantee eager diffusion, using the formal definition presented in Section 5.4.

Though the proof of correctness of IO Systems is rather long, there are just a few key ideas. One that deserves repetition is the notion of a return pair (Definition 35). Return pairs describe the conditions under which the *cmt-closure* operation, which represents descendant inference, is valid. We belive that this notion could be useful in other proofs as well. As we acknowledged in Chapter 1, the idea of inferring commits based upon the absence of aborts is not original in our protocol and seems to be an interesting technique that could be applicable under other circumstances as well.

We would also like to reemphasize that IO Systems do not completely model the distribution of information involved in the actual protocol, though appropriate modifications of the definitions should not be difficult. There are two issues to be dealt with. The first issue involves the granularity of commit and abort information that is stored. In this sense IO Systems are more "distributed" than our real systems because they treat each object and transaction as a separate entity. Commit and abort information is accessible per object and transaction (by unioning the information stored for all operations of a given object or transaction), rather than per collection of objects and transactions, as is true in the real protocol where information is stored for each guardian. The second issue involves the locality of the information used in computations. In this sense, IO Systems are less distributed than our real systems because they make use of a global set of created actions in the computation of inferences defined by *known-vis*.

It seems clear that enlarging the storage granularity of commit and abort information will not invalidate the correctness of IO Systems. The non-determinism already present in the protocol allows extra information to be added to committed and aborted sets. The postconditions on all operations ensure that only correct information is added to these sets and that when commit information is added, enough abort information is also added to maintain return pairs. Thus only minor changes should be required to introduce the notion of a "guardian" as a collection of transactions and objects.

In order to obtain a truly distributed model of the protocol we must also ensure that the controller's state information may be partitioned among guardians such that no guardian attempts to compute with non-local information. This is almost possible with the current definitions, but there is one problem. Using the current definition of *known-vis*, a guardian would need access to a global created set to compute the inferences about commits that can be made for a local access transaction. By introducing one additional assumption on generic objects, we can change the definition of *known-vis* to use only local information about created transactions. We need to assume that the only transactions whose commits will matter at a site are the ancestors of access transactions that ran at the site. (It is not difficult to be convinced

of this for locking objects, at least). Under this assumption, the only creates that must be considered in computing *known-vis* are those for local access transactions and their ancestors; this information is available locally at the guardian that computes it. Based on this we can define a modified IO System that, under the assumption about generic objects, uses the "local" definition of *known-vis* to guarantee that transactions see the same thing that they see in IO Systems. Thus we would be proving a weak simulation of IO Systems by the modified IO systems.

Chapter 7

Conclusion

7.1 Summary

Existing nested action commit protocols have problems both in efficiency and semantics. The efficiency of nested action completion in the different protocols varies widely; some provide reasonable performance—though there is still room for improvement—while others have proved to be completely impractical. Most of the protocols provide relatively weak guarantees about when sites learn the outcomes of actions. In this thesis we have presented a design and correctness proof for a new nested action commit protocol, addressing issues of semantics and efficiency.

We introduced the notion of an *eager diffusion semantics* for action completion. This semantics is contrasted with the *lazy diffusion semantics* supported by most existing nested action commit protocols. In a system that uses locking for concurrency control, eager diffusion guarantees that whenever an action knows that it ought to be able to obtain a lock, perhaps because it has observed the abort of the action that it knows last held the lock, then the site will be able to grant the lock based on purely local information. An action's knowledge includes its state, the programs of potentially all other actions in the system, and deductions that can be made from this knowledge. A site's local information consists of the states of local locks and explicit information recorded at the site about commits and aborts of actions. A site does not interpret an action's state or program, hence the possibility of a discrepancy between a site's information and an action's knowledge. In contrast to eager diffusion, a system that supports lazy diffusion guarantees only that sites learn the outcomes of actions eventually, with no time bound. In such a system, a site may require communication with other sites in order to grant a lock request even if the requesting action already knows that the lock should be available.

A simplistic design for a protocol to support eager diffusion could be very expensive. By piggybacking information about commits and aborts on messages already flowing around the system, and by devising a series of optimizations based on inference mechanisms and efficient data structures, we developed a protocol that we believe may be practical. Our rigorous correctness proof formalizes the definition of

eager diffusion and makes explicit the assumptions needed to ensure the correctness of the inference mechanisms.

Some of the optimizations we employ in our protocol are generalizations of *ad hoc* methods employed in existing nested action commit protocols. Not surprisingly, these optimizations can be applied in their more general form to improve the performance of existing protocols. In particular, we examined how some of our optimizations could be used in the Argus nested action commit protocol to reduce the number of lock propagation query messages that must be generated when a lock is requested.

While our optimizations go a long way in limiting the amount of information that must be propagated on messages to support eager diffusion, there are still some problems. The greatest expense in our protocol involves the storage and garbage collection of information about topactions. Haphazard elimination of some or all of this information results in a protocol that cannot support eager diffusion. We have described one possible garbage collection scheme for our protocol, but there are still questions about whether it will be efficient enough to use in practice.

In applying our optimizations to improve the performance of existing protocols, garbage collection does not pose a serious problem. In those cases we are interested in producing protocols that still support only lazy diffusion but which propagate commit and abort information faster than a protocol that is entirely driven by queries. Work in implementation and performance analysis will help to reveal which additional optimizations are worthwhile in such protocols. The evaluation also depends, to a large extent, on the types of applications that run in the system and the kinds of lock conflicts that arise. Further experience in building applications should help us to understand the tradeoffs better.

7.2 Future Work

This thesis is far from an exhaustive study of commit protocols for nested actions. There are a number of unresolved issues concerning the work we have presented, as well as a number of other worthwhile avenues of pursuit in this area.

The primary question that remains in our work is whether the protocol will be useful in practice. This depends to a large extent upon whether our garbage collection scheme will work, and, if not, whether more efficient schemes can be devised. Another interesting question is whether ideas from our protocol will prove useful in improving the efficiency of protocols that support a weaker semantics.

Additional work on the formal model and proof of our protocol is also warranted. Since working through the version of the proof presented in this thesis, we have identified points where additional modularization could simplify the statements and proofs of the lemmas. In particular, the formal proof could be made to follow more closely the breakdown that we used in our informal correctness arguments. Rather than proving the correctness of both inferencing mechanisms together, as we did in

proving that the Inference Optimized Controller simulates the Local Unoptimized Controller in Chapter 6, we could prove the correctness of descendant inferences and sequential inferences independently of one another. Thus, we would break the IOC into two parts, proving that the first part simulates the LUC and that the second part simulates the first. Additionally, we would like to prove correct a variant of the IOC that uses only to local information about created actions for making inferences. We discussed this in Section 6.4.

Other open questions in the formal part of our work involve proving the correctness of the protocol in a system that models crashes, proving that the protocol combines correctly with abort and crash orphan detection algorithms, and proving liveness properties for the controllers.

We decided to limit our study to systems that use locking for concurrency control. The question of how to handle commits and aborts of nested actions in systems that do not use locking is still wide open. What do we do when there are no lock requests to drive queries? Must such systems introduce significant delays at each level of the action tree if they are to support an eager diffusion semantics? What exactly does eager diffusion mean in these systems? Also, there are still open questions concerning the meaning of eager diffusion with respect to user-defined atomic types.

We also limited our study to a particular model of nested actions where individual actions are not distributed and the action management, communication, and recovery functions are all integrated in one module (a guardian). In the Camelot and Clouds systems, for example, these functions are more loosely coupled. The question of what changes would be required to adapt the protocol for such systems remains open. Additional questions arise in adapting the protocol for Camelot because action identifiers in that system contain less information about the history of actions; we rely upon the easy availability of history information in our inference mechanisms.

Appendix A

Postponed Lemmas and Proofs

This appendix contains the proofs and lemmas that were ommitted from Chapter 6 in the interests of readability.

A.1 Postponed Proofs

A.1.1 Return Pairs

Lemma 37. Let α be a well-formed IOC schedule, γ an IOC execution such that $schedule(\gamma) = \alpha$, ϕ an operation in α, s' a state immediately preceding the last instance of ϕ in γ or any time after that, $\langle A, R \rangle$ a return pair for ϕ in s', s the state immediately after γ, $A' \subseteq \text{aborted}(s)$. Then

$$\forall T \in desc(R).\ anc(T) \cap A' \cap desc(R) \neq \emptyset \Longrightarrow anc(T) \cap A \cap desc(R) \neq \emptyset.$$

Proof: Pick any $U \in R$, $T \in desc(U)$. We must show:

$$anc(T) \cap A' \cap desc(U) \neq \emptyset \Longrightarrow anc(T) \cap A \cap desc(U) \neq \emptyset.$$

By definition of return pair, we know:

F1. $\forall T \in (R - A).\text{COMMIT}(T, v)$ occurs before some instance of ϕ in α, and

F2. $\forall T \in R.\forall T' \in desc(T).$
$T' \in \text{aborted}(s') \Longrightarrow anc(T') \cap desc(T) \cap A \neq \emptyset.$

If $U \in A$ then the result is immediate. So assume $U \notin A$. By F1 then, $\psi = \text{COMMIT}(U, v)$ occurs before some instance of ϕ in α. Let U' be an ancestor of T that is a descendant of U in A'. By Lemma 25, U is in committed(s). By Lemma 32 and the fact that $A' \subseteq \text{aborted}(s)$, U' must in fact be a proper descendant of U. Let s'' be the state just before ψ in γ. By Lemma 43,

$\forall T' \in prop\text{-}desc(U) \cap \text{create-requested}(s'').$
either $anc(T') \cap prop\text{-}desc(U) \cap \text{aborted}(s'') \neq \emptyset$
or $anc(T') \cap prop\text{-}desc(U) \subseteq \text{committed}(s'')$.

By F2,

$$\forall T' \in desc(U).T' \in \text{aborted}(s') \Longrightarrow anc(T') \cap desc(U) \cap A \neq \emptyset \ .$$

By Lemma 31, committed(s'') $\subseteq$ committed(s) and aborted(s'') $\subseteq$ aborted(s'). Thus,

$$\begin{aligned}&\forall T' \in prop\text{-}desc(U) \cap \text{create-requested}(s'').\\ &\quad \text{either } anc(T') \cap desc(U) \cap A \neq \emptyset\\ &\quad \text{or} \quad anc(T') \cap prop\text{-}desc(U) \subseteq \text{committed}(s) \ .\end{aligned}$$

In particular, this holds for T. Now, the second alternative cannot be true because U' cannot be in both committed(s) and aborted(s) by Lemma 32. So it must be that $anc(T) \cap desc(U) \cap A \neq \emptyset$. Thus the result holds. ■

A.1.2 Return Pairs for Sequential Inferences

Lemma 38. Let α be a well-formed IOC schedule, $\phi = \text{COMMIT}(T, v) \in \alpha$, and s a state of the IOC during α anytime after ϕ. Then

$$\begin{aligned}&\forall T' \in prop\text{-}desc(T).T' \in \text{aborted}(s) \Longrightarrow\\ &\quad anc(T') \cap \text{abort-map}(s)(\phi) \cap prop\text{-}desc(T) \neq \emptyset \ .\end{aligned}$$

Proof: Let T' be any transaction in $prop\text{-}desc(T)$. The proof is by induction on the number of transactions between T' and T. Let γ be an IOC execution containing s such that $schedule(\gamma) = \alpha$. Let $\alpha'\phi$ be a prefix of α, and let s' be the state immediately after $\alpha'\phi$ in γ. The basis is that the number of transactions between T' and T is 0. In this case, T' is a child of T. $T' \in \text{aborted}(s')$ implies that $\text{ABORT}(T')$ occurs in α' (since the COMMIT does not happen until all request-created children have returned). By Lemma 30, $T' \in \text{abort-map}(s')(\text{ABORT}(T'))$. By the preconditions on COMMIT requiring a REQUEST-COMMIT to have occurred, the postconditions on REQUEST-COMMIT and COMMIT and Lemma 31, $T' \in \text{abort-map}(s)(\phi)$ and the claim is true.

For the induction hypothesis, assume the claim is true when the number of transactions between T' and T is $k-1$. We will show that the claim is true when the number of transactions is k. Let T'' be the ancestor of T' that is a child of T. A return operation for T'' must occur before ϕ in α' by the controller preconditions on COMMIT requiring a REQUEST-COMMIT to have occurred and the restrictions on REQUEST-COMMITs of transactions. Let ψ be the return operation and let s'' be the state immediately after ψ in γ. If $\psi = \text{ABORT}(T'')$ then, by the postconditions on ABORT, $T'' \in \text{abort-map}(s'')(\text{ABORT}(T''))$ and by the postconditions on REQUEST-COMMIT and COMMIT $T'' \in \text{abort-map}(s')(\phi)$. Thus $T'' \in \text{abort-map}(s)(\phi)$ also, by Lemma 31, and we're done. If $\psi = \text{COMMIT}(T'', v')$ then there are $k-1$ transactions between T'' and T' and by the induction hypothesis, $anc(T') \cap \text{abort-map}(s'')(\psi) \cap prop\text{-}desc(T'') \neq \emptyset$. By the IOC postconditions on REQUEST-COMMIT and COMMIT

and Lemma 31 abort-map(s'')(ψ) $\subseteq$ abort-map(s')(ϕ) $\subseteq$ abort-map(s)(ϕ), so the claim holds. ■

Lemma 39. Let α be a well-formed IOC schedule, s a state of the IOC during α, ϕ any operation, and $T \in$ *seq-infer*(s, ϕ). Then a return operation for T prefix-affects all instances of ϕ in α.

Proof: First, note that if $\phi \notin \alpha$ or s is not a state during α sometime after the first instance of ϕ then there is nothing to prove since *seq-infer*$(s, \phi) = \emptyset$. So assume s is a state of the IOC anytime after the first instance of ϕ. Consider any $T \in$ *seq-infer*(s, ϕ). $T \in$ created(s) by definition. We will examine the possibilities for ϕ. If $\phi =$ REQUEST-COMMIT(T', v), where $T' =$ *parent*(T), then ϕ is directly-prefix-affected by a return operation for T since a REQUEST-COMMIT cannot occur until all request-created children have returned (by the restriction on transactions). If $\phi =$ COMMIT(T', v) or $\phi =$ INFORM-COMMIT-AT(X)OF(T') then by transitivity all instances of ϕ are prefix-affected by the return for T. This is because REQUEST-COMMIT(T', v) directly-prefix-affects COMMIT(T', v) which directly-prefix-affects INFORM-COMMIT-AT(X)OF(T') (and in a well-formed schedule, the REQUEST-COMMIT must precede the COMMIT which must precede all instances of the INFORM-COMMIT).

If T_ϕ is a descendant of a later sequential sibling of T then let T' be the ancestor of T_ϕ that is a sibling of T. The REQUEST-CREATE for T' must follow the return for T by definition of "later sequential sibling". Thus the return for T directly-prefix-affects REQUEST-CREATE(T') in α. It is easy to see from the definition of the prefix-affects relation that a REQUEST-CREATE of a transaction prefix-affects the operations of its descendants, so by transitivity, the return for T prefix-affects all instances of ϕ in α. ■

Lemma 40. Let α be an IOC schedule, γ an IOC execution such that *schedule*$(\gamma) = \alpha$, ψ, ϕ in α such that ψ is a return operation that prefix-affects an instance of ϕ in α, s a state any time after the instance of ϕ prefix-affected by ψ in γ, and s' the state immediately after ψ in γ. Then abort-map(s')(ψ) $\subseteq$ abort-map(s)(ϕ).

Proof: By induction on the length of a prefix-affects chain between ψ and ϕ. Let s'' be the state immediately after an instance of ϕ prefix-affected by ψ in γ. In the basis, ψ directly-prefix-affects ϕ in α. Then it is easy to see from the IOC postconditions on each operation that abort-map(s')(ψ) $\subseteq$ abort-map(s'')(ϕ). By Lemma 31, abort-map(s'')(ϕ) $\subseteq$ abort-map(s)(ϕ) and the result follows by transitivity.

For the induction hypothesis assume that the claim is true for any prefix-affects chain of length $\leq n - 1$. We will show that it is true for any prefix-affects chain of length n. Fix some prefix-affects chain between ψ and ϕ of length n. By a property

of transitive closure, if ψ does not directly-prefix-affect ϕ in α then there exists a π on the prefix-affects chain between ψ and ϕ such that π directly-prefix-affects φ in α and ψ prefix-affects π in α. It is easy to see that there exists a prefix-affects chain between ψ and π of length $n-1$ (namely the one between ψ and ϕ with φ omitted). So by the induction hypothesis abort-map$(s')(\psi) \subseteq$ abort-map$(s''')(\pi)$. where s''' is the state immediately after π in γ. Also by the induction hypothesis, abort-map$(s''')(\pi) \subseteq$ abort-map$(s)(\phi)$ since π directly-prefix-affects ϕ in α. By transitivity then abort-map$(s')(\psi) \subseteq$ abort-map$(s)(\phi)$. ■

Lemma 41. Let α be a well-formed IOC schedule, ϕ any operation, s a state during α, $T \in$ *seq-infer*(s, ϕ). Then

$$\forall T' \in \mathit{desc}(T).T' \in \text{aborted}(s) \Longrightarrow \mathit{anc}(T') \cap \mathit{desc}(T) \cap \text{abort-map}(s)(\phi) \neq \emptyset$$

Proof: Let γ be an IOC execution containing s such that *schedule*$(\gamma) = \alpha$. By Lemma 39, a return operation for T prefix-affects all instances of ϕ in α. Call this operation ψ and let s' be the state immediately after ψ in γ. We know that state s must occur sometime after the first instance of ϕ since *seq-infer*$(s, \phi) \neq \emptyset$. So by Lemma 40, abort-map$(s')(\psi) \subseteq$ abort-map$(s)(\phi)$. Now there are two cases. First consider $\psi =$ ABORT(T). By controller postconditions on ABORT, $T \in$ abort-map$(s')(\psi)$ so $T \in$ abort-map$(s)(\phi)$ and the claim holds. Now consider $\psi =$ COMMIT(T, v). By Lemma 38, $\forall T' \in$ *prop-desc*$(T).T' \in$ aborted$(s) \Longrightarrow$ *anc*$(T') \cap$ *prop-desc*$(T) \cap$ abort-map$(s)(\psi) \neq \emptyset$. Since ψ may only occur once in α, abort-map$(s)(\psi) =$ abort-map$(s')(\psi) \subseteq$ abort-map$(s)(\phi)$. Also, $T \notin$ aborted(s) by the precondition on COMMIT and Lemmas 31 and 32. Thus the claim holds. ■

A.1.3 Return Pairs for Cmap Inferences

Lemma 43. Let α be a well-formed IOC schedule, $\phi =$ COMMIT(T, v) and operation in α, and s' a state of the IOC immediately before ϕ. Then

$$\begin{aligned}&\forall T' \in \mathit{prop\text{-}desc}(T) \cap \text{create-requested}(s').\\ &\quad \text{either } \mathit{anc}(T') \cap \mathit{prop\text{-}desc}(T) \cap \text{aborted}(s') \neq \emptyset, \text{ or}\\ &\qquad \mathit{anc}(T') \cap \mathit{prop\text{-}desc}(T) \subseteq \text{committed}(s') \ .\end{aligned}$$

Proof: Let $T' \in$ *prop-desc*$(T) \cap$ create-requested(s'). The proof is by induction on the number of transactions between T' and T. The basis is when the number of transactions is 0. In this case, T' is a child of T. Since ϕ cannot occur until all its request-created children have returned, a return for T' must occur before ϕ. So either $T' \in$ aborted(s') or $T' \in$ committed(s') by the controller postconditions on COMMIT and ABORT and Lemma 31.

For the induction hypothesis, assume that the claim holds when there are $k-1$ transactions between T' and T. We'll show that it is true when the number of

transactions is k. Let $T'' = parent(T')$. There are at most $k-1$ transactions between T'' and T so by the induction hypothesis either $anc(T'') \cap prop\text{-}desc(T) \cap \text{aborted}(s') \neq \emptyset$ or $anc(T'') \cap prop\text{-}desc(T) \subseteq \text{committed}(s')$. If the first case is true, then also $anc(T') \cap prop\text{-}desc(T) \cap \text{aborted}(s') \neq \emptyset$ and we are done. If the second case is true then, in particular, $T'' \in \text{committed}(s')$, so $\text{COMMIT}(T'', v')$ must occur before ϕ in α by Lemma 28. Let γ be an IOC execution containing s' such that $schedule(\gamma) = \alpha$. Let s'' be the state immediately before $\text{COMMIT}(T'', v')$ in γ. There are at most $k-1$ transactions between T'' and T', so by the induction hypothesis, either $anc(T') \cap prop\text{-}desc(T'') \cap \text{aborted}(s'') \neq \emptyset$ or $anc(T') \cap prop\text{-}desc(T'') \subseteq \text{committed}(s'')$. If the former is true then the result follows from Lemma 31. If the latter is true then it, combined with the fact that $anc(T'') \cap prop\text{-}desc(T) \subseteq \text{committed}(s')$ and Lemma 31, gives the result. ■

Lemma 44. Let α be a well-formed IOC schedule, ϕ any operation, s a state during α, $T \in conc\text{-}infer(s, \phi)$. Then

$$\forall T' \in desc(T).T' \in \text{aborted}(s) \Longrightarrow anc(T') \cap desc(T) \cap \text{abort-map}(s)(\phi) \neq \emptyset$$

Proof: Let γ be an IOC execution containing s such that $schedule(\gamma) = \alpha$. Note that ϕ must occur in α and s must be a state sometime after the first instance of ϕ in α for T to exist. Let π be the last instance of ϕ in γ before s. Let s' be the state immediately after π. Since the cmap for ϕ only changes when ϕ occurs, $T \in conc\text{-}infer(s', \phi)$ also. Let T' be a descendant of T in aborted(s). In fact, T' must be a proper descendant of T by Lemmas 27 and 32. Further, a $\text{COMMIT}(T, v)$ must occur before s' in γ by Lemmas 27 and 28. Let s'' be the state immediately after $\text{COMMIT}(T, v)$ in γ. By Lemmas 43, 32, and 31, T' has an ancestor T'' below T in aborted(s''). By Lemma 31, $T'' \in$ aborted(s'). By the fifth IOC postcondition on all operations, T'' itself has an ancestor below T in abort-map(s')(ϕ). Thus, by Lemma 31, T' has an ancestor below T in abort-map(s)(ϕ). ■

A.1.4 Commit Closures

Lemma 49. Let C_1, C_2, A_1, A_2, R_1, R_2 be sets of transactions such that the following conditions hold:

C1. $C_1 \subseteq C_2$, and

C2. R_2 dominates R_1, and

C3. $anc\text{-}between(R_1, R_2) \cap A_2 = \emptyset$, and

C4. $\forall T \in desc(R_1).\, anc(T) \cap A_2 \cap desc(R_1) \neq \emptyset \Longrightarrow anc(T) \cap A_1 \cap desc(R_1) \neq \emptyset$.

Then $cmt\text{-}closure(C_1, A_1, R_1) \subseteq cmt\text{-}closure(C_2, A_2, R_2)$.

Proof: By Lemma 64 and C4,

$$cmt\text{-}closure(C_1, A_1, R_1) \subseteq cmt\text{-}closure(C_1, A_2, R_1).$$

By Lemma 62, C2, and C3,

$$cmt\text{-}closure(C_1, A_2, R_1) \subseteq cmt\text{-}closure(C_1, A_2, R_2).$$

By Lemma 63 and C1,

$$cmt\text{-}closure(C_1, A_2, R_2) \subseteq cmt\text{-}closure(C_2, A_2, R_2).$$

Thus the result holds by transitivity of $\subseteq$. ■

A.2 Postponed Lemmas

A.2.1 Lemmas About Commit-Closures

Lemma 61. Let C, A, R_1, R_2 be sets of transactions. Then

$$cmt\text{-}closure(C, A, R_1) \cup cmt\text{-}closure(C, A, R_2) = cmt\text{-}closure(C, A, (R_1 \cup R_2)).$$

Proof: Easy from definitions. ■

Lemma 62. Let C, A, R_1, R_2 be sets of transactions such that R_2 dominates R_1 and $anc\text{-}between(R_1, R_2) \cap A = \emptyset$. Then $cmt\text{-}closure(C, A, R_1) \subseteq cmt\text{-}closure(C, A, R_2)$.

Proof: Expanding the definition of *cmt-closure*, we want to show

$$C \cap desc(R_1) \cap non\text{-}orphans(A, R_1) \subseteq C \cap desc(R_2) \cap non\text{-}orphans(A, R_2).$$

Consider any $T \in C \cap desc(R_1)$. Since R_2 dominates R_1, $T \in C \cap desc(R_2)$ also. $T \in non\text{-}orphans(A, R_1)$ means $anc(T) \cap desc(R_1) \cap A = \emptyset$. What we need to show is that $anc(T) \cap desc(R_2) \cap A = \emptyset$. It is not hard to see that

$$anc(T) \cap desc(R_2) = (anc(T) \cap desc(R_1)) \cup (anc(T) \cap anc\text{-}between(R_1, R_2)).$$

Thus, by the condition of the lemma, $anc(T) \cap desc(R_2) \cap A = \emptyset$, as required. ■

Lemma 63. Let C_1, C_2, A, R be sets of transactions such that $C_1 \subseteq C_2$. Then $cmt\text{-}closure(C_1, A, R) \subseteq cmt\text{-}closure(C_2, A, R)$.

Proof: Easy from definitions. ■

Lemma 64. Let C, A_1, A_2, R be sets of transactions such that: $\forall T \in desc(R).\, anc(T) \cap A_2 \cap desc(R) \neq \emptyset \Longrightarrow anc(T) \cap A_1 \cap desc(R) \neq \emptyset$. Then $cmt\text{-}closure(C, A_1, R) \subseteq cmt\text{-}closure(C, A_2, R)$.

Proof: Expanding the definition of *cmt-closure*, we want to show:

$$C \cap desc(R) \cap non\text{-}orphans(A_1, R) \subseteq C \cap desc(R) \cap non\text{-}orphans(A_2, R).$$

Consider any $T \in C \cap desc(R)$. $T \in non\text{-}orphans(A_1, R)$ means $anc(T) \cap desc(R) \cap A_1 = \emptyset$. By the contrapositive of the condition of the lemma then, we get $anc(T) \cap A_2 \cap desc(R) = \emptyset$. Thus $T \in non\text{-}orphans(A_2, R)$ as required. ■

A.2.2 Miscellaneous Lemmas

Lemma 65. Let α be an IOC schedule, ϕ any operation, s a state of the IOC during α. Then $seq\text{-}infer(s, \phi)$ is a mutually unrelated set.

Proof: Easy from definitions. ■

Lemma 66. Let α be an IOC schedule, ϕ any operation, s a state of the IOC during α. Then $conc\text{-}infer(s, \phi)$ is a mutually unrelated set.

Proof: Easy from definitions. ■

Lemma 67. Let $\mathcal{S}$ and $\mathcal{T}$ be sets of transactions, where $\mathcal{T}$ is mutually unrelated and $\mathcal{S} \subseteq \mathcal{T}$. Then $anc\text{-}between(\mathcal{S}, \mathcal{T}) = \emptyset$.

Proof: Easy from definitions. ■

Lemma 68. Let T, T' be transactions such that T' is an ancestor of T, and $\mathcal{S}$ a set of transactions containing children of those ancestors of T that are descendants of T'. Then any transaction in $anc\text{-}between(\mathcal{S}, \{T'\})$ is an ancestor of T that is also a descendant of T'.

Proof: Easy from definitions. ■

Lemma 69. Let $\mathcal{S}$ and $\mathcal{T}$ be sets of transactions where $\mathcal{S}$ is a mutually unrelated set, and $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2$. Then

$$anc\text{-}between(\mathcal{S}, \mathcal{T}) = anc\text{-}between(\mathcal{S}_1, \mathcal{T}) \cup anc\text{-}between(\mathcal{S}_2, \mathcal{T}).$$

Proof: Note that when $\mathcal{S}$ is mutually unrelated, $anc\text{-}between(\mathcal{S}, \mathcal{T}) = prop\text{-}anc(S) \cap desc(T)$. The rest is easy from the definitions. ■

Lemma 70. Let α be an IOC schedule, π a REQUEST-COMMIT operation in α for an access to object X, ϕ an operation at X in α such that an instance of ϕ occurs before π and T_ϕ is locally visible to T_π at X in the prefix of α ending at π, and s a state sometime after π during α. Then all ancestors of T_ϕ that are proper descendants of $lca(T_\phi, T_\pi)$ are in committed(s).

Proof: By definition, T_ϕ locally visible to T_π at X in the prefix of α ending in π means that INFORM-COMMIT operations have occurred at X before π for each ancestor of T_ϕ that is a proper descendant of $lca(T_\phi, T_\pi)$. Obviously then, by the preconditions on INFORM-COMMIT which require a COMMIT to have occurred, and Lemma 25, these transactions must be in the committed set by the time π occurs in α. ■

Lemma 71. Let α be an IOC schedule, s a state of the IOC during α, and ϕ any operation. Then abort-map(s)(ϕ) $\cap$ committed(s) = $\emptyset$.

Proof: If s does not follow some instance of ϕ then the result is immediately obvious. Otherwise, let γ be an IOC execution containing s such that $schedule(\gamma) = \alpha$. Let s' be the state in γ immediately after the last instance of ϕ that precedes s. By Lemma 29, abort-map(s')(ϕ) = abort-map(s)(ϕ). By the fourth IOC postcondition on all operations, abort-map(s')(ϕ) $\subseteq$ aborted(s'). Then the result follows by Lemma 32. ■

Lemma 72. Let T be any transaction, T'' a proper ancestor of T, T' the child of T'' that is an ancestor of T, and $\mathcal{S}$ a set of children of ancestors of T. Then $\mathcal{S} = ca(T'', \mathcal{S}) \cup cd(T', \mathcal{S})$.

Proof: Easy from definitions. ■

Corollary 73. Let T be any transaction, T'' the parent of T, and $\mathcal{S}$ a set of children of proper ancestors of T. Then $\mathcal{S} = ca(T'', \mathcal{S})$.

Proof: Easy from definitions. ■

Lemma 74. Let $\alpha\pi$ be an IOC schedule, γ an IOC execution such that $schedule(\gamma) = \alpha\pi$, ϕ an operation in α such that an instance of ϕ occurs before an instance of π in $\alpha\pi$, s' the state immediately after α in γ, s the state immediately after $\alpha\pi$ in γ. Let $T'' = lca(T_\phi, T_\pi)$. Then $ca(T'', seq\text{-}infer(s', \phi)) \subseteq ca(T'', seq\text{-}infer(s, \pi))$.

Proof: Easy from definitions. ■

Lemma 75. Let α be an IOC schedule, ϕ some operation, s a state of the IOC during α. If $seq\text{-}infer(s, \phi)$ contains children of T_ϕ and $location(\phi) = parent(T_\phi)$, then

1. ϕ is a COMMIT operation, and
2. $T_\phi \in$ committed(s).

Proof: Looking at the definition of *seq-infer*(s, ϕ), it will only contain descendants of T_ϕ if ϕ is one of REQUEST-COMMIT, COMMIT, or INFORM-COMMIT. Since $location(\phi) = parent(T_\phi)$, ϕ must be a COMMIT. If *seq-infer*(s, ϕ) is non-empty, s must be a state sometime after ϕ in α. Thus, by Lemma 25, T_ϕ must be in committed(s). ■

Lemma 76. Let α be an IOC schedule. γ an IOC execution such that *schedule*$(\gamma) = \alpha$, ϕ, π operations, s', s states in γ, $T'' = lca(T_\phi, T_\pi)$. Then,

$$conc\text{-}with(T_\pi, cpa(T_\pi, \text{cmap}(s')(\phi))) \subseteq \text{cmap}(s)(\pi) \Longrightarrow$$
$$cpa(T'', conc\text{-}infer(s', \phi)) \subseteq cpa(T'', conc\text{-}infer(s, \pi)) .$$

Proof: The proof is by straightforward manipulation of the definitions. We need to show

$$cpa(T'', conc\text{-}with(T_\phi, cpa(T_\phi, \text{cmap}(s')(\phi))))$$
$$\subseteq cpa(T'', conc\text{-}with(T_\pi, cpa(T_\pi, \text{cmap}(s)(\pi)))) .$$

First we notice that *cpa* and *conc-with* commute to give

$$cpa(T'', cpa(T_\pi, conc\text{-}with(T_\pi, \text{cmap}(s)(\phi)))) .$$

This reduces to

$$cpa(T'', conc\text{-}with(T_\phi, \text{cmap}(s')(\phi))) .$$

Then this is equivalent to

$$cpa(T'', conc\text{-}with(T'', \text{cmap}(s')(\phi))) \subseteq cpa(T'', conc\text{-}with(T'', \text{cmap}(s)(\pi))) .$$

From the condition of the lemma we can get

$$cpa(T'', conc\text{-}with(T'', \text{cmap}(s')(\phi))) \subseteq \text{cmap}(s)(\pi) .$$

This implies the desired result. ■

Lemma 77. Let T be any transaction, T'' a proper ancestor of T, T' the child of T'' that is an ancestor of T, $\mathcal{S}$ a set of children of proper ancestors of T. Then $cd(T'', \mathcal{S}) = c(T'', \mathcal{S}) \cup cd(T', \mathcal{S})$.

Proof: Easy from definitions. ■

Corollary 78. Let T be any transaction, T'' the parent of T, and $\mathcal{S}$ a set of children of proper ancestors of T. Then $cd(T'', \mathcal{S}) = c(T'', \mathcal{S})$.

Proof: Easy from definitions. ■

Lemma 79. Let $\alpha\pi$ be an IOC schedule, γ an IOC execution such that $schedule(\gamma) = \alpha\pi$, ϕ an operation in α such that an instance of ϕ occurs before an instance of π and T_ϕ and T_π are descendants of sequential siblings, s' the state immediately after α in γ, s the state immediately after $\alpha\pi$ in γ, and $T'' = lca(T_\phi, T_\pi)$. Then $c(T'', conc\text{-}infer(s', \phi)) \subseteq c(T'', seq\text{-}infer(s, \pi))$.

Proof: Let U and U' be the ancestors of T_ϕ and T_π, respectively, that are children of T''. All transactions in $c(T'', conc\text{-}infer(s', \phi))$ are concurrent siblings of U. Since U is a prior sequential sibling of U', these must all be prior sequential siblings of U' also. Since $seq\text{-}infer(s, \pi)$ contains all created prior sequential siblings of U', the result follows immediately. ■

Lemma 80. For any operation π and IOC state s, $seq\text{-}infer(s, \pi)$ is a set of children of ancestors of T_π.

Proof: Easy from definition of *seq-infer*. ■

Lemma 81. Let T be any transaction, T'' a proper ancestor of T, T' the child of T'' that is an ancestor of T, $\mathcal{S}$ a set of children of ancestors of T. Then $cpd(T'', \mathcal{S}) = cd(T', \mathcal{S})$.

Proof: Easy from definitions. ■

Corollary 82. Let T be any transaction, T'' the parent of T, and $\mathcal{S}$ a set of children of ancestors of T. Then $cpd(T'', \mathcal{S}) = c(T, \mathcal{S})$.

Proof: Easy from definitions. ■

Lemma 83. Let α be an IOC schedule, γ an IOC execution such that $schedule(\gamma) = \alpha, \phi, \pi$ operations in α such that T_ϕ and T_π are descendants of concurrent siblings, s' and s states in γ, $T'' = lca(T_\phi, T_\pi)$. If $conc\text{-}with(T_\pi, cpa(T_\pi, \mathrm{cmap}(s')(\phi))) \subseteq \mathrm{cmap}(s)(\pi)$ then $ca(T'', conc\text{-}infer(s, \phi)) \subseteq conc\text{-}infer(s, \pi)$.

Proof: The proof is by straightforward manipulation of the definitions. We want to show

$$ca(T'', conc\text{-}with(T_\phi, cpa(T_\phi, \mathrm{cmap}(s')(\phi)))) \subseteq conc\text{-}with(T_\pi, cpa(T_\pi, \mathrm{cmap}(s)(\pi))) .$$

Now we can decompose the left-hand side into

$$cpa(T'', conc\text{-}with(T_\phi, cpa(T_\phi, \mathrm{cmap}(s')(\phi))))$$
$$\cup\, c(T'', conc\text{-}with(T_\phi, cpa(T_\phi, \mathrm{cmap}(s')(\phi)))) .$$

This reduces to

$$cpa(T'', conc\text{-}with(T'', \mathrm{cmap}(s')(\phi))) \cup c(T'', conc\text{-}with(T_\phi, \mathrm{cmap}(s')(\phi))) .$$

The first set is a subset of $conc\text{-}infer(s, \pi)$ by Lemma 76. Since T_ϕ and T_π are concurrent, the second set is a subset of

$$c(T'', conc\text{-}with(T_\pi, \mathrm{cmap}(s')(\phi))) ,$$

which by the condition of the lemma, is a subset of $\mathrm{cmap}(s)(\pi)$. The result follows straightforwardly from there. ■

Lemma 84. Let $\alpha\pi$ be an IOC schedule, ϕ an operation in α, where π and ϕ are operations at some object X such that an instance of ϕ occurs before an instance of π and T_ϕ and T_π are descendants of concurrent siblings, s a state of the IOC immediately after $\alpha\pi$, T_ϕ locally visible to T_π at X in the prefix of α ending with the last instance of π, π an output operation, and T the ancestor of T_ϕ that is a child of $lca(T_\phi, T_\pi)$. Then $T \in conc\text{-}infer(s, \pi)$.

Proof: Let γ be an IOC execution containing s such that $schedule(\gamma) = \alpha\pi$. T_ϕ locally visible to T_π at X in the prefix of α ending with the last instance of π means that for every ancestor T' of T_ϕ below $lca(T_\phi, T_\pi)$, $\psi = \text{INFORM-COMMIT-AT}(X)\text{OF}(T')$ has occurred at X before some instance of π. In particular, this is true for T. Let s' be the state just before the last instance of π in γ. By Lemma 33, $T \in \mathrm{cmap}(s')(\psi)$. Since π is an output operation at X it must be REQUEST-COMMIT. By the controller postconditions for π, $conc\text{-}with(T_\pi, cpa(T_\pi, \mathrm{cmap}(s')(\psi))) \subseteq \mathrm{cmap}(s)(\pi)$. So $T \in \mathrm{cmap}(s)(\pi)$. Then it is easy to see from the definition of *conc-infer* that $T \in conc\text{-}infer(s, \pi)$. ■

Lemma 85. Let α be an IOC schedule, ϕ a COMMIT operation and π a REQUEST-CREATE operation such that $location(\phi) = location(\pi) = T$, ϕ occurs before π and T_ϕ and T_π are concurrent siblings, and s a state of the IOC sometime after π. Then $T_\phi \in c(T, conc\text{-}infer(s, \pi))$.

Proof: Let γ be an IOC execution containing s such that $schedule(\gamma) = \alpha$. Let s' be the state just before π in γ. By Lemma 33, $T_\phi \in \mathrm{cmap}(s')(\phi)$. By the postconditions on π, $conc\text{-}with(T_\pi, cpa(T_\pi, \mathrm{cmap}(s')(\phi))) \subseteq \mathrm{cmap}(s)(\pi)$. So $T_\phi \in \mathrm{cmap}(s)(\pi)$. Then it is easy to see from the definition of *conc-infer* that $T_\phi \in c(T, conc\text{-}infer(s, \pi))$. ■

Appendix B

Modelling the Abort Set Optimization

In this appendix we define Aborts Optimized Systems to model the optimization from Chapter 4 that allows transactions to be replaced by their ancestors in aborts sets. Aborts Optimized Systems are composed of transactions, generic objects (that satisfy a particular condition) and the Aborts Optimized Controller, which we define in Section B.1. As a result of this exercise, we conclude that Aborts Optimized Systems weakly simulate Inference Optimized Systems, in that schedules of an AO System look the same as schedules of the corresponding IO System to non-access transactions. This weak simulation result is not quite sufficient to prove that Aborts Optimized Systems guarantee eager diffusion. In the last section we discuss this further.

B.1 Aborts Optimized Controller

Notation: In the postconditions below $S \preceq T$ means S *is dominated by* T.

All states of the Aborts Optimized Controller (AOC) are the same as those of the IOC, except that abort-map is called amap to emphasize that it contains possibly different information than the IOC's abort-map.

As usual, the initial state is denoted by s_0, and all sets are initially empty in s_0 except for create-requested, which is $\{T_0\}$. The functions $\text{cmap}(s_0)$ and $\text{amap}(s_0)$ map each operation to the empty set. As usual, define $\textit{returned}(s) = \text{committed}(s) \cup \text{aborted}(s)$.

In the transition relations for the operations of the AOC defined below, every operation π contains the following additional postconditions:

1. $\text{cmap}(s')(\pi) \subseteq \text{cmap}(s)(\pi)$
2. $\text{cmap}(s)(\pi) \subseteq \text{committed}(s)$
3. $\text{amap}(s')(\pi) \preceq \text{amap}(s)(\pi)$
4. $\text{amap}(s)(\pi) \subseteq \text{aborted}(s)$
5. $\forall T \in \text{cmap}(s)(\pi). \forall T' \in \text{aborted}(s) \cap \textit{prop-desc}(T).\ \textit{anc}(T') \cap \text{amap}(s)(\pi) \neq \emptyset$

All other state components remain the same from s' to s unless explicitly mentioned in the postconditions below.

- π = REQUEST-CREATE(T)

  Post: create-requested(s) = create-requested(s') $\cup \{T\}$
  $\forall \phi.\mathit{location}(\phi) = \mathit{parent}(T) \Longrightarrow$
  $\mathit{conc\text{-}with}(T, \mathit{cpa}(T, \text{cmap}(s')(\phi))) \subseteq \text{cmap}(s)(\pi)$ and
  $\text{amap}(s')(\phi) \preceq \text{amap}(s)(\pi)$

- π = REQUEST-COMMIT(T, v)

  Post: commit-requested(s) = commit-requested(s') $\cup \{(T, v)\}$
  $\forall \phi.\mathit{location}(\phi) = \mathit{location}(\pi) \Longrightarrow$
  $\mathit{conc\text{-}with}(T, \mathit{cpa}(T, \text{cmap}(s')(\phi))) \subseteq \text{cmap}(s)(\pi)$ and
  $\text{amap}(s')(\phi) \preceq \text{amap}(s)(\pi)$

- π = CREATE(T)

  Pre: $T \in$ create-requested(s') $-$ created(s')
  if T is an access to X then
  $\mathit{known\text{-}vis}(s', \text{REQUEST-CREATE}(T)) \subseteq$ informed-commit(s')(X) and
  $\text{amap}(s')(\text{REQUEST-CREATE}(T)) \subseteq$ informed-abort(s')(X)
  Post: created(s) = created(s') $\cup \{T\}$
  $\text{cmap}(s')(\text{REQUEST-CREATE}(T)) \subseteq \text{cmap}(s)(\pi)$
  $\text{amap}(s')(\text{REQUEST-CREATE}(T)) \preceq \text{amap}(s)(\pi)$

- π = COMMIT(T, v)

  Pre: $(T, v) \in$ commit-requested(s')
  $T \notin \mathit{returned}(s')$
  Post: committed(s) = committed(s') $\cup \{T\}$
  $\text{cmap}(s')(\text{REQUEST-COMMIT}(T, v)) \subseteq \text{cmap}(s)(\pi)$
  $\text{amap}(s')(\text{REQUEST-COMMIT}(T, v)) \preceq \text{amap}(s)(\pi)$
  if $\mathit{concurrent}(T)$ then $T \in \text{cmap}(s)(\pi)$

- π = ABORT(T)

  Pre: $T \in$ create-requested(s') $-$ $\mathit{returned}(s')$
  Post: aborted(s) = aborted(s') $\cup \{T\}$
  $\text{cmap}(s')(\text{REQUEST-CREATE}(T)) \subseteq \text{cmap}(s)(\pi)$
  $\text{amap}(s')(\text{REQUEST-CREATE}(T)) \cup \{T\} \preceq \text{amap}(s)(\pi)$

- π = INFORM-COMMIT-AT(X)OF(T)

  Pre: $T \in$ committed(s')
  Post: $\forall v.(T,v) \in$ commit-requested(s') $\Longrightarrow$
  cmap(s')(COMMIT(T,v)) $\subseteq$ cmap(s)(π) and
  amap(s')(COMMIT(T,v)) $\preceq$ amap(s)(π)
  informed-commit(s)(X) = informed-commit(s')(X) $\cup$ $\{T\}$

- π = INFORM-ABORT-AT(X)OF(T)

  Pre: $T \in$ aborted(s')
  Post: cmap(s')(ABORT(T)) $\subseteq$ cmap(s)(π)
  amap(s')(ABORT(T)) $\preceq$ amap(s)(π)
  informed-abort(s)(X) = informed-abort(s')(X) $\cup$ $\{T\}$

This controller differs from the IOC in that when ϕ prefix-affects π we now require only that for every transaction in ϕ's amap, π's amap contain an ancestor of the transaction. The fifth postcondition on all operations also changes so that we now require that there be some ancestor in the amap for every aborted descendant of a transaction in the corresponding cmap. The IOC had the additional requirement that this ancestor be a proper descendant of the transaction in the cmap.

B.2 Generic Object Assumption

In order to prove that Inference Optimized Systems look the same as Aborts Optimized Systems to non-access transactions (*i.e.*, the weak simulation result) we need to impose a restriction on generic objects (in addition to Assumption 16, which we also assume here). The reason that AO Systems do not strongly simulate IO Systems is that, upon creating an access, the AOC does not have as much local information as the IOC has. The generic object assumption says, in effect, that we still have enough information. When we replace transactions by their ancestors in the amap we lose the information that those particular transactions aborted. What we still know is that the effects of those transactions must be undone, since their ancestors' aborts will prevent them from ever committing to the top in any case.

The generic object assumption requires that, if an object is informed of the abort of a transaction and later is informed of the abort of one of the transaction's descendants. then the later INFORM-ABORT has no affect on the behavior of the object. The notion of *equieffectiveness* of schedules [Fekete *et al.* 1988] is used to formalize the requirement.

Definition 86. Let α and β be schedules of some automaton $\mathcal{A}$, and γ a sequence of operations of $\mathcal{A}$ such that $\alpha\gamma$ and $\beta\gamma$ are well-formed. Then α is equieffective to β, written $\alpha \cong \beta$, if $\alpha\gamma$ a schedule of $\mathcal{A} \Longleftrightarrow \beta\gamma$ a schedule of $\mathcal{A}$.

Lemma 87. Let α, β, and γ be sequences of operations of some automaton $\mathcal{A}$, where $\alpha \cong \beta$ and $\beta \cong \gamma$. If $\forall\delta.(\alpha\delta$ well-formed and $\gamma\delta$ well-formed) $\Longrightarrow$ $\beta\delta$ well-formed then $\alpha \cong \gamma$.

Proof: Easy from definition of equieffectiveness. ■

Assumption 88. Let α be a well-formed schedule of a generic object X, where INFORM-ABORT-AT(X)OF(T) $\in \alpha$, T' is a descendant of T, and π = INFORM-ABORT-AT(X)OF(T'). Then $\alpha\pi \cong \alpha$.

Lemma 89. Let α be a well-formed schedule of a generic object X, and δ a sequence of INFORM-ABORT-AT(X) operations for transactions in a set $\mathcal{T}$ such that $\forall T \in \mathcal{T}.$(INFORM-COMMIT-AT($X$)OF($T$) $\notin \alpha$ and $\exists T' \in anc(T)$.INFORM-ABORT-AT(X)OF(T') $\in \alpha$). Then $\alpha\delta$ is well-formed and $\alpha\delta \cong \alpha$.

Proof: By induction on the length of δ. For the base case, let δ = INFORM-ABORT-AT(X)OF(T). It is easy to see that $\alpha\delta$ is well-formed if α is well-formed, since an INFORM-ABORT operation for a transaction T may occur at an object at any time and any number of times as long as no INFORM-COMMIT operation for the transaction has occcurred. By the generic object assumption then, $\alpha \cong \alpha\delta$.

For the induction hypothesis, assume that the claim is true for δ length $< k$. We'll show that it is true for δ of length k. Let $\delta = \delta'\pi$ be of length k, where π is a single INFORM-ABORT-AT(X) operation. By the induction hypothesis, $\alpha\delta'$ is well-formed and $\alpha \cong \alpha\delta'$. Also by the induction hypothesis, $\alpha\delta' \cong \alpha\delta'\pi$. Since $\delta\pi$ only contains INFORM-ABORTs, and there are no INFORM-COMMITs for the same transactions, it does not affect other generic object operations with regards to well-formedness, so $\forall\gamma.(\alpha\gamma$ well-formed and $\alpha\delta'\pi\gamma$ well-formed) $\Longrightarrow$ $\alpha\delta'\gamma$ well-formed. Thus by Lemma 87, $\alpha \cong \alpha\delta$. ■

The following is a simple extension lemma for equieffective schedules.

Lemma 90. Let α, β and γ be sequences of operations of an automaton A, such that $\alpha\gamma$ and $\beta\gamma$ are well-formed schedules of $\mathcal{A}$ and $\alpha \cong \beta$. Then $\alpha\gamma \cong \beta\gamma$.

Proof: Easy from definition of equieffectiveness. ■

B.3 Aborts Optimized Systems

Aborts Optimized Systems are composed of transactions, generic objects that satisfy the generic object assumption, and the Aborts Optimized Controller.

B.4 Simulation of IO Systems by AO Systems

The weak simulation result of this section says that every AO schedule looks the same to non-access transactions as an IO schedule.

For this simulation result, we explicitly define the mapping between states of the AOC and states of the IOC (in the previous section, the mapping was simpler and thus left implicit in the lemmas).

Definition 91. The predicate *aoc-ioc-map*(s, t) relates states s of the AOC to states t of the IOC as follows:

aoc-ioc-map(s, t) =
$\forall \pi$. abort-map$(t)(\pi)$ = *desc*(amap$(s)(\pi)) \cap$ aborted(s), and
$\forall X$. informed-abort$(t)(X) \supseteq$ informed-abort$(s)(X)$, and
All other state components of t = the corresponding components of s.

The following are two technical lemmas used in the proof of Lemma 94 to show that certain postconditions of the IOC hold under the state mapping.

Lemma 92. Let $\mathcal{S}$, $\mathcal{S}'$, $\mathcal{T}$, $\mathcal{T}'$, and $\mathcal{A}$ be sets of transactions such that $\mathcal{T} = desc(\mathcal{S}) \cap \mathcal{A}$, $\mathcal{T}' = desc(\mathcal{S}') \cap \mathcal{A}$ and $\mathcal{S} \preceq \mathcal{S}'$. Then $\mathcal{T} \subseteq \mathcal{T}'$.

Proof: Consider any $T \in \mathcal{T}$. We want to show that $T \in \mathcal{T}'$. $T \in desc(\mathcal{S}) \cap \mathcal{A}$ by assumption. Since $\mathcal{S} \preceq \mathcal{S}'$, and T is a descendant of an action in $\mathcal{S}$, T must be a descendant of an action in $\mathcal{S}'$ also. Thus $T \in desc(\mathcal{S}') \cap \mathcal{A}$. Thus $T \in \mathcal{T}'$, as required. ■

Lemma 93. Let T, T' be transactions, and $\mathcal{S}$, $\mathcal{T}$, and $\mathcal{A}$ be sets of transactions such that $T' \in$ *prop-desc*$(T) \cap \mathcal{A}$, $\mathcal{T} = desc(\mathcal{S}) \cap \mathcal{A}$, and $anc(T') \cap \mathcal{S} \neq \emptyset$. Then $anc(T') \cap \mathcal{T} \cap$ *prop-desc*$(T) \neq \emptyset$.

Proof: Since $T' \in \mathcal{A}$ and has an ancestor in $\mathcal{S}$, $T' \in \mathcal{T}$ also. Then, since T' is a proper descendant of T, it is easy to see that $T' \in anc(T') \cap \mathcal{T} \cap$ *prop-desc*(T). Thus the claim is true. ■

The next lemma states that if the AOC and the IOC are in corresponding states with an operation π enabled for both controllers, then after both execute π, there will be a next state of the IOC satisfying the state mapping for any next state of the AOC.

Lemma 94. Let (s', π, s) be a step of the AOC, t' a state of the IOC such that *aoc-ioc-map*(s', t') = true and π is enabled in state t'. Then there exists a state t of the IOC such that (t', π, t) is a step of the IOC and *aoc-ioc-map*(s, t) = true.

Proof: Let t be a state of the IOC as follows:

$\forall \pi$. abort-map$(t)(\pi) = \mathit{desc}(\text{amap}(s)(\pi)) \cap \text{aborted}(s)$, and
if $\pi =$ INFORM-ABORT-AT(X)OF(T) then
    informed-abort$(t)(X) =$ informed-abort$(t')(X) \cup \{T\}$, and
for all other X, informed-abort$(t)(X) =$ informed-abort$(t')(X)$, and
All other components of t are equal to the corresponding components of s.

We must show that the IOC postconditions are satisfied in state t and that $\mathit{aoc\text{-}ioc\text{-}map}(s,t) =$ true.

First we will show that the state mapping holds. For all components other than informed-abort, the claim follows immediately from the definitions of t and the state mapping. Now, if π is not an INFORM-ABORT then by definition of t, $\forall X$. informed-abort$(t)(X) =$ informed-abort$(t')(X)$. By the state mapping, informed-abort$(t')(X) \supseteq$ informed-abort$(s')(X)$. And by inspection of the AOC, informed-abort$(s')(X) =$ informed-abort$(s)(X)$. Thus informed-abort$(t)(X) \supseteq$ informed-abort$(s)(X)$, as required. If $\pi =$ INFORM-ABORT-AT(X)OF(T) then for $X' \neq X$, informed-abort$(t)(X') \supseteq$ informed-abort$(s)(X')$ for the same reason as above. For $X' = X$, informed-abort$(t')(X) \supseteq$ informed-abort$(s')(X)$ by the state mapping. By the definition of t, informed-abort$(t)(X) =$ informed-abort$(t')(X) \cup \{T\}$. By the AOC postcondition on π, informed-abort$(s)(X) =$ informed-abort$(s')(X) \cup \{T\}$. Thus informed-abort$(t)(X) \supseteq$ informed-abort$(s)(X)$, as required.

Now we will show that the IOC postconditions after π are satisfied in state t. First consider the five IOC postconditions that apply to all π. The first two conditions clearly hold since the cmap and committed components of s and t are equal and the conditions hold for s by the first two AOC postconditions on all π. The third condition holds by Lemma 92 and the third AOC postcondition on all π. The fourth holds by the definition of the state mapping since $\forall \pi$. abort-map$(t)(\pi) \subseteq$ aborted(s) and aborted$(s) =$ aborted(t). The fifth condition holds by Lemma 93 and the fifth AOC postcondition on all π.

Now consider the postconditions on individual operations, π. Clearly, the IOC postconditions involving the components that are the same in s and t (all except abort-map and informed-abort) hold for t since they hold for s by the AOC postconditions on π. The IOC postconditions on abort-map hold by Lemma 92 and the AOC postconditions on π. When $\pi =$ INFORM-ABORT-AT(X)OF(T) the IOC postconditions on π hold by definition of t. ■

Now we want to prove a lemma relating *known-vis* in the AOC to *known-vis* in the IOC. Notice that $\mathit{known\text{-}vis}(s,\pi)$ in the AOC is not necessarily a subset of committed(s). If $\mathit{seq\text{-}infer}(s,\pi)$ or $\mathit{conc\text{-}infer}(s,\pi)$ contains a transaction that has a proper ancestor in amap$(s)(\pi)$ (and thus is an orphan), then that orphaned transaction and its descendants could appear in $\mathit{known\text{-}vis}(s,\pi)$ but not really be committed. This could happen because the *cmt-closure* function only checks for aborted ances-

tors up through transactions in the set over which the closure is being taken. If the aborted descendants of an transaction in the set are represented in the abort set by an ancestor above that transaction, *cmt-closure* will not notice it. What this means in the execution of the controller is that the controller could block on a CREATE operation that has this kind of *known-vis* set. The CREATE would never become enabled since it would be waiting for INFORM-COMMITs that could never occur. However, it is easy to see that any such CREATE is an operation of an orphan itself, and thus we do not care if the operation is never enabled. (Since the *seq-infer* and *conc-infer* sets contain only children of proper ancestors of the transaction being created, if transactions in those sets have proper ancestors in the amap then so does the transaction being created). In the actual protocol, we will not "block" but instead make a possibly incorrect inference. This inference would only be seen by a transaction that is an orphan. However we cannot then claim that the eager diffusion semantics is guaranteed for orphan transactions. Note that the orphan is detectable as such based upon information in the aborted set. Thus we could detect and kill the orphan before it sees any incorrect inferences.

Lemma 96 says that *known-vis* in a state of the AOC during some execution contains strictly more information that *known-vis* in a corresponding state of the IOC. To prove this we first prove one more small lemma about a property of *cmt-closure*.

Lemma 95. Let C, A_1, A_2, and R be sets of transactions, where $A_2 \subseteq A_1$. Then $\mathit{cmt\text{-}closure}(C, A_1, R) \subseteq \mathit{cmt\text{-}closure}(C, A_2, R)$.

Proof: Expanding the definition of *cmt-closure*, we need to show

$$C \cap \mathit{desc}(R) \cap \mathit{non\text{-}orphans}(A_1, R) \subseteq C \cap \mathit{desc}(R) \cap \mathit{non\text{-}orphans}(A_2, R).$$

Looking at the definition of *non-orphans* it is clear that when $A_2 \subseteq A_1$ that $\mathit{non\text{-}orphans}(A_1, R) \subseteq \mathit{non\text{-}orphans}(A_2, R)$. Thus the claim holds. ■

Lemma 96. Let s be a state of the AOC such that $\forall \pi.\, \mathrm{amap}(s)(\pi) \subseteq \mathrm{aborted}(s)$, t a state of the IOC such that $\mathit{aoc\text{-}ioc\text{-}map}(s,t) = \mathrm{true}$, π an operation such that $\mathit{has\text{-}occurred}(s,\pi) = \mathrm{true}$. Then $\mathit{known\text{-}vis}(t,\pi) \subseteq \mathit{known\text{-}vis}(s,\pi)$.

Proof: We will show that

$$\begin{aligned}&\mathit{cmt\text{-}closure}(\mathrm{created}(t), \mathrm{abort\text{-}map}(t)(\pi), \mathit{seq\text{-}infer}(t,\pi)) \\ &\qquad \subseteq \mathit{cmt\text{-}closure}(\mathrm{created}(s), \mathrm{amap}(s)(\pi), \mathit{seq\text{-}infer}(s,\pi)),\end{aligned}$$

and that the same holds if *conc-infer* is substituted for *seq-infer*. This then implies the desired result.

It is easy to see from the definitions that $\textit{seq-infer}(t,\pi) = \textit{seq-infer}(s,\pi)$ and $\textit{conc-infer}(t,\pi) = \textit{conc-infer}(s,\pi)$. By the state mapping, abort-map$(t)(\pi) = \textit{desc}(\text{amap}(s)(\pi)) \cap \text{aborted}(s)$. Since amap$(s)(\pi) \subseteq$ aborted(s), abort-map$(t)(\pi) \supseteq$ amap$(s)(\pi)$. Also, by the state mapping, created(s) = created(t). The result then follows by Lemma 95 and the definition of *known-vis*. ■

Our last theorem says that the AO System looks the same to every non-access transaction as the corresponding IO System, in that anything a transaction can see in a schedule of the AO System it could see in some schedule of the IO System. In fact, we prove that given any AO Schedule, there is a single IO Schedule that looks the same to every non-access transaction.

Theorem 97. The AOC simulates the IOC in the following sense. Let α be a schedule of the AO System (AOC, transactions, generic objects) and let the generic object assumption hold for all generic objects. Then there exists a schedule β of the IO System (IOC, same transactions and generic objects) such that: $\forall T$ non-access transactions.$\alpha \mid T = \beta \mid T$, and $\forall X$ generic objects.$\beta \mid X \cong \alpha \mid X$.

Proof: By induction on the length of AO System schedules. What we will actually do is construct a sequence of IO System operations β such that for α an AO System schedule, and s a state of the AOC after α:

1. β is a schedule of the IO System, and
2. $\forall T.\alpha \mid T = \beta \mid T$, and
3. $\forall X.\beta \mid X \cong \alpha \mid X$, and
4. $\exists t$ a state of the IOC after β.*aoc-ioc-map*(s,t), and
5. β differs from α only in that β may contain extra INFORM-ABORT operations for transactions in aborted(s).

The base case, α of length 0 is trivial. For the induction hypothesis, assume the claim is true for α of length $< k$. We'll show that the claim holds for α of length k.

Let $\alpha = \alpha'\pi$ be an AO System schedule of length k, where π is a single operation. Let s' be a state of the AOC after α' such that (s',π,s) is a step of the AOC. By the induction hypothesis, there exists an IO System schedule β' such that $\forall T.\alpha' \mid T = \beta' \mid T$, $\forall X.\beta' \mid X \cong \alpha' \mid X$, and $\exists t'$ a state of the IOC after β'.*aoc-ioc-map*(s',t'). Now we'll consider cases for π depending upon which system component outputs π.

- π is an output of a transaction T
  Let $\beta = \beta'\pi$.
  1. By assumption, $\alpha'\pi$ is a schedule of the AO System, so $\alpha'\pi \mid T$ is a schedule of T. By the induction hypothesis, $\beta' \mid T = \alpha' \mid T$. So clearly, $\beta'\pi \mid T$ is a schedule of T. Also by the induction hypothesis, β' is a schedule of the IO System. Thus, by Lemma 1, $\beta'\pi$ is a schedule of the IO System.

2. It follows easily from the induction hypothesis that $\forall T'.\alpha'\pi \mid T' = \beta'\pi \mid T'$.
3. Since π is not an operation of any object, and $\forall X.\beta' \mid X \cong \alpha' \mid X$, clearly $\forall X.\beta'\pi \mid, X \cong \alpha'\pi \mid X$.
4. By assumption, (s', π, s) is a step of the AOC. By the induction hypothesis, $aoc\text{-}ioc\text{-}map(s', t') =$ true. Since π is an output of a transaction, it is an input to the IOC so clearly π is enabled in t'. Thus, by Lemma 94, there exists a state t of the IOC after $\beta'\pi.aoc\text{-}ioc\text{-}map(s, t)$.
5. Immediate from induction hypothesis.

- π is an output of a generic object X.
  Let $\beta = \beta'\pi$.
  1. By assumption, $\alpha'\pi$ is a schedule of the AO System, so $\alpha'\pi \mid X$ must be a well-formed schedule of X. The induction hypothesis implies that if $\alpha'\pi \mid X$ and $\beta'\pi \mid X$ are well-formed sequences of generic object operations, and $\alpha'\pi \mid X$ is a schedule of X then $\beta'\pi \mid X$ is a schedule of X (by the definition of equieffective). Since $\alpha'\pi$ is a schedule of the AO System, $\alpha'\pi \mid X$ is a well-formed sequence of generic object operations. By the induction hypothesis, $\beta'\pi \mid X$ differs from $\alpha'\pi \mid X$ only in that the former may contain extra INFORM-ABORT operations for transactions in aborted(s'). By the state mapping, aborted(t') = aborted(s'). Since no INFORM-COMMIT operation may appear occur for a transaction that is aborted, these extra INFORM-ABORTs do not affect well-formedness for sequences of generic object operations, and $\beta'\pi \mid X$ must also be a well-formed sequence of generic object operations. Thus $\beta'\pi \mid X$ is a schedule of X. Then by Lemma 1, $\beta'\pi$ is a schedule of the IO System.
  2. It follows easily from the induction hypothesis that $\forall T.\alpha'\pi \mid T = \beta'\pi \mid T$.
  3. By the induction hypothesis, $\forall X'.\beta' \mid X' \cong \alpha' \mid X'$. For $X' \neq X$, it is easy to see that $\beta'\pi \mid X' \cong \alpha'\pi \mid X'$. For $X' = X$, we proved in 1 above that $\beta'\pi \mid X$ and $\alpha'\pi \mid X$ are schedules of X. Thus by Lemma 90, $\beta'\pi \mid X \cong \alpha'\pi \mid X$.
  4. By assumption, (s', π, s) is a step of the AOC. By the induction hypothesis, $aoc\text{-}ioc\text{-}map(s', t') =$ true. Since π is an output of a generic object, it is an input to the IOC, so π is enabled in t'. Thus by Lemma 94, there exists a state t of the IOC after $\beta'\pi$ such that $aoc\text{-}ioc\text{-}map(s, t) =$ true.
  5. Immediate by induction hypothesis.
- π is an output of the AOC but $\pi \neq$ CREATE(T) for T an access.
  Let $\beta = \beta'\pi$.
  1. By the induction hypothesis, β' is a schedule of the IO System, and so is also a schedule of the IOC. It is easy to see from the definition of the state mapping and inspection of the AOC and IOC preconditions that since π is enabled in state s' then π is enabled in state t'. Thus $\beta'\pi$ is a schedule of the

IOC and, by Lemma 1 it is also a schedule of the IO System.

2. It follows easily from the induction hypothesis that $\forall T.\alpha'\pi \mid T = \beta'\pi \mid T$.
3. By the induction hypothesis, $\forall X.\beta' \mid X \cong \alpha' \mid X$. If π is not an operation of X, it is easy to see that $\beta'\pi \mid X \cong \alpha'\pi \mid X$. If π is an operation of an X, then since $\beta'\pi$ is a schedule of the IO System, $\beta'\pi \mid X$ must be a schedule of X. Also, since $\alpha'\pi$ is a schedule of the AO System then $\alpha'\pi \mid X$ must be a schedule of X. Thus by Lemma 90, $\beta'\pi \mid Xn \cong \alpha'\pi \mid X$.
4. By assumption, (s', π, s) is a step of the AOC. By the induction hypothesis, $aoc\text{-}ioc\text{-}map(s', t') =$ true. By 1 above, π is enabled in t'. Thus by Lemma 94, there exists a state t of the IOC after $\beta'\pi$ such that $aoc\text{-}ioc\text{-}map(s, t) =$ true.
5. Immediate by induction hypothesis.

- $\pi =$ CREATE(T) where T is an access to X

  Let $\beta = \beta'\delta\pi$ where δ is a sequence of INFORM-ABORT-AT(X)OF(T') operations, one for each $T' \in$ (abort-map(t')(REQUEST-CREATE(T)) $-$ amap(s')(REQUEST-CREATE(T))). Let t'' be a state of the IOC after $\beta'\delta$ that is reachable from t' by executing δ. (We will show in step 1 that t'' exists.)

  1. By the induction hypothesis, β' is an IO System schedule and hence an IOC schedule. For every $\phi \in \delta$, $T_\phi \in$ aborted(t') since $\forall\pi.$ abort-map(t')(π) $\subseteq$ aborted(t') by the IOC postconditions on all π and the monotonicity of the aborted set. Thus, after β', each operation in δ is enabled in turn and $\beta'\delta$ is an IOC schedule (and so t'' exists). Looking at the IOC postconditions on INFORM-ABORT t'' simply has an informed-abort set that is a superset of t''s. Thus since $aoc\text{-}ioc\text{-}map(s', t') =$ true, $aoc\text{-}ioc\text{-}map(s', t'') =$ true also. Now we need to show that π is enabled in t''. For brevity, let $\phi =$ REQUEST-CREATE(T). By Lemma 96, $known\text{-}vis(t'', \phi) \subseteq known\text{-}vis(s', \phi)$. (The Lemma applies because, by well-formedness, $has\text{-}occurred(s', \phi)$ must be true if π is enabled in s'. It is easy to see from the state mapping that $has\text{-}occurred(t'', \phi)$ must also be true then. The condition on amap(s') holds by the AOC postconditions on all operations and the monotonicity of the aborted set.). Since π is enabled in s', $known\text{-}vis(s', \phi) \subseteq$ informed-commit(s')(X). By the state mapping, informed-commit(s') = informed-commit(t''). Thus $known\text{-}vis(t'', \phi) \subseteq$ informed-commit(t''). By the AOC preconditions on CREATE amap(s')(ϕ) $\subseteq$ informed-abort(s')(X). Thus, by the state mapping, amap(s')(ϕ) $\subseteq$ informed-abort(t'')(X). By construction, abort-map(t'')(ϕ) $-$ amap(s')(ϕ) $\subseteq$ informed-abort(t'')(X). So abort-map(t'')(ϕ) $\subseteq$ informed-abort(t'')(X) and π is enabled in t''. Since $\beta'\delta$ is an IOC schedule, then $\beta'\delta\pi$ is an IOC schedule and also an IO System schedule by Lemma 1.
  2. It follows easily from the induction hypothesis that $\forall T.\alpha'\pi \mid T = \beta'\delta\pi \mid T$ since $\delta\pi$ contains no operations of T.

3. $\forall X' \neq X.(\alpha' \mid X' = \alpha'\pi \mid X'$ and $\beta' \mid X' = \beta'\delta\pi \mid X')$ so $\forall X' \neq X.\alpha'\pi \mid X' \cong \beta'\delta\pi \mid X'$.
   By the induction hypothesis, $aoc\text{-}ioc\text{-}map(s', t') =$ true, so
   abort-map(t')(REQUEST-CREATE(T)) =
   $desc($amap(s')(REQUEST-CREATE$(T)) \cap$ aborted(s') .
   Thus amap(s')(REQUEST-CREATE(T)) contains an ancestor for every transaction in abort-map(t')(REQUEST-CREATE(T)). By the AOC preconditions for CREATE, INFORM-ABORT-AT(X)OF(T') occurs in α' for each $T' \in$ amap(s')(REQUEST-CREATE(T)). By the induction hypothesis, β' also contains these INFORM-ABORTs. All INFORM-ABORTs in δ will be for descendants of transactions in amap(s')(REQUEST-CREATE(T)). Then by Lemma 89, $\beta' \mid X \cong \beta'\delta \mid X$. By the induction hypothesis, $\alpha' \mid X \cong \beta' \mid X$. Since δ contains only INFORM-ABORTs for aborted actions, it is clear that $\forall\gamma.(\alpha'\gamma \mid X$ well-formed and $\beta'\delta\gamma$ well-formed$) \Longrightarrow \beta'\gamma$ well-formed. Thus, by Lemma 87, $\alpha' \mid X \cong \beta'\delta \mid X$. By assumption, $\alpha'\pi \mid X$ is a schedule of X. Since $\beta'\delta\pi$ is an IO System schedule (by 1 above), $\beta'\delta\pi \mid X$ is a schedule of X. Thus by Lemma 87, $\alpha'\pi \mid X \cong \beta'\delta\pi \mid X$.
4. By assumption, (s', π, s) is a step of the AOC. As shown in 1 above, $aoc\text{-}ioc\text{-}map(s', t'') =$ true and π is enabled in t''. Thus by Lemma 94, there exists a state t of the IOC after $\beta'\delta\pi$ such that $aoc\text{-}ioc\text{-}map(s, t) =$ true.
5. This follows by the induction hypothesis, and the fact that abort-map(t')(REQUEST-CREATE(T)) $\subseteq$ aborted(t').

■

B.5 AO Systems and Eager Diffusion

It turns out that the result stated in Theorem 97 is not quite what is needed to prove that AO Systems guarantee eager diffusion. Here we develop a notion of *equiinformative* sequences of operations, and indicate what must be done to prove that AO Systems guarantee eager diffusion using this notion. While we have not worked out all the details of a proof that AO Systems guarantee eager diffusion, we have worked out enough that we believe the rest to be straightforward.

Definition 98. Let α and β be sequences of generic operations. We say that α is *equiinformative* to β if α and β differ only in that β may contain additional operations INFORM-ABORT-AT(X)OF(T), where $\exists T' \in anc(T)$.INFORM-ABORT-AT(X)OF(T') occurs previously in β and also occurs in α.

Lemma 99. Let α and β be generic schedules, $T, U \in accesses(X)$, and $\alpha \mid X$ be equiinformative to $\beta \mid X$. Then $\beta \vdash \mathcal{L}(U, T, X) \Longrightarrow \alpha \vdash \mathcal{L}(U, T, X)$.

Proof:

$\beta \vdash \mathcal{L}(U,T,X) \stackrel{\text{def}}{\equiv} \text{CREATE}(U) \in \beta \Longrightarrow$
(U is locally-visible at X to T in β) or
($\exists U' \in anc(U).\text{INFORM-ABORT-AT}(X)\text{OF}(U') \in \beta$).

Assume CREATE(U) $\in \beta$. Note that CREATE(U) $\in \alpha$ also by the equiinformative assumption. If U is locally-visible at X to T in β then U is locally-visible at X to T in α since α and β contain the same INFORM-COMMITs at X. Otherwise, let $U' \in anc(U)$ such that INFORM-ABORT-AT(X)OF(U') $\in \beta$. If INFORM-ABORT-AT(X)OF(U') $\in \alpha$ then we're done. Otherwise, by the equiinformative assumption, $\exists U'' \in anc(U').\text{INFORM-ABORT-AT}(X)\text{OF}(U'') \in \alpha$. Since $U'' \in anc(U)$ also, we're done. ■

Theorem 100. Let S be an implementation of a generic system such that:

$\forall$ schedules α of $S.\exists$ a GKS schedule β.
$(\forall T.\alpha \mid T = \beta \mid T)$ and $(\forall X.\alpha \mid X$ is equiinformative to $\beta \mid X)$.

Then, S satisfies eager diffusion.

Proof: We want to show:

$\forall$ schedules α of S ending in CREATE(T), for $T \in accesses(X)$ and T is not orphaned in α.
$\forall U \in accesses(X).\alpha \vdash K_T(\mathcal{G}(U,T)) \Longrightarrow \alpha \vdash \mathcal{L}(U,T,X)$.

The plan is as follows. Fix a particular α and β. Let β' be the prefix of β ending in CREATE(T). (We can easily show that β' exists from the hypotheses of the theorem.) We will show:

P1. $\beta' \mid T = \alpha \mid T$,

P2. $\forall U \in accesses(X).\alpha \vdash K_T(\mathcal{G}(U,T)) \Longrightarrow \beta' \vdash K_T(\mathcal{G}(U,T))$, and

P3. $\forall U \in accesses(X).\beta' \vdash \mathcal{L}(U,T,X) \Longrightarrow \alpha \vdash \mathcal{L}(U,T,X)$.

Then we can conclude that $\forall U \in accesses(X).\alpha \vdash K_T(\mathcal{G}(U,T)) \Longrightarrow \alpha \vdash \mathcal{L}(U,T,X)$ based on the result of Theorem 18.

P1: By well-formedness, $\beta' \mid T = \alpha \mid T = \text{CREATE}(T)$.

P2: Fix any $U \in accesses(X)$. $\alpha \vdash K_T(\mathcal{G}(U,T)) \stackrel{\text{def}}{\equiv} \forall$ generic schedules $\gamma.\gamma \mid T = \alpha \mid T \Longrightarrow \gamma \vdash \mathcal{G}(U,T)$.
Since $\beta' \mid T = \alpha \mid T$, $\{\gamma : \gamma \mid T = \alpha \mid T\} = \{\gamma : \gamma \mid T = \beta' \mid T\}$.
The result then follows by substitution.

P3: If we can show that $\alpha \mid X$ is equiinformative to $\beta' \mid X$ then Lemma 99 gives the result. Let $\beta = \beta'\delta$. Since $\alpha \mid X$ is equiinformative to $\beta \mid X$, and α ends in CREATE(T), $\delta \mid X$ can only contain INFORM-ABORT-AT(X) operations for transactions U' such that α contains an INFORM-ABORT-AT(X)OF(U'') for some $U'' \in anc(U')$. Since α contains the INFORM-ABORT for the ancestor, so does β'. Thus $\alpha \mid X$ is equiinformative to $\beta \mid X$.

■

It remains to prove that AO Systems have the property required in the statement of Theorem 100. While this does not follow directly from Theorem 97, we can modify that theorem by replacing $\cong$ with *equiinformative* in the statement, and by modifiying the proof to include the equiinformativeness requirement in the induction hypothesis.

References

[Allchin 1983] James E. Allchin. *An Architecture for Reliable Decentralized Systems.* Technical Report GIT-ICS-83/23, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia, September 1983.

[Aspnes 1987] J. Aspnes. *Timestamp Ordering and Nested Transactions.* Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, June 1987.

[Birman & Joseph 1987] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987. Also available as TR 85-694, Dept. of Computer Science, Cornell University, Ithaca, NY.

[Duchamp & Spector 1987] Dan Duchamp and Alfred Spector. *Handling Orphans in Camelot Release 1.0.* Camelot Working Memo 15, Dept. of Computer Science, Carnegie Mellon University, Pittsburgh, PA, September 1987. Draft.

[Duchamp 1987] Dan Duchamp. October 1987. Private communication.

[Dwork & Skeen 1983] Cynthia Dwork and Dale Skeen. The inherent cost of nonblocking commitment. In *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing*, pages 1–11, ACM, August 1983.

[Fekete *et al.* 1987] Alan Fekete, Nancy Lynch, Michael Merritt, and William Weihl. Nested transactions and read/write locking. In *Proceedings of the 6th ACM Symposium on Principles of Database Systems*, pages 97–111, ACM, 1987. An expanded version is available as Technical Memo 324, MIT Laboratory for Computer Science, Cambridge, MA, April 1987.

[Fekete *et al.* 1988] Alan Fekete, Nancy Lynch, Michael Merritt, and William Weihl. *Commutativity-Based Locking for Nested Transactions.* Technical Memo 370, MIT Laboratory for Computer Science, Cambridge, MA, August 1988.

[Gray 1979] J.N. Gray. Notes on database operating systems. In R. Bayer, R.M. Graham, and G. Seegmüller, editors, *Operating Systems: An Advanced Course*, chapter 3.F, pages 394–481, Springer-Verlag, 1979.

[Gray *et al.* 1976] J. N. Gray, R. A. Lorie, G. F. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. In G. M. Nijssen, editor, *Modelling in Data Base Management Systems*, North Holland, 1976.

[Greif *et al.* 1987] Irene Greif, Robert Seliger, and William Weihl. *A Case Study of CES: A Distributed Collaborative Editing System Implemented in Argus.* Programming Methodology Group Memo 55, MIT Laboratory for Computer Science, Cambridge, MA, April 1987. To appear in IEEE Transactions on Software Engineering.

[Halpern & Moses 1987] Joseph Y. Halpern and Yoram Moses. *Knowledge and Common Knowledge in a Distributed Environment.* Research Report RJ4421, IBM Almaden Research Center, San Jose, CA, August 1987. Revised version. Original version October, 1984.

[Herlihy *et al.* 1987] Maurice Herlihy, Nancy Lynch, Michael Merritt, and William Weihl. *On the Correctness of Orphan Elimination Algorithms.* Technical Memo 329, MIT Laboratory for Computer Science, Cambridge, MA, May 1987. Accepted for publication in Journal of the ACM.

[Joseph & Birman 1986] Thomas A. Joseph and Kenneth P. Birman. Low cost management of replicated data in fault-tolerant distributed systems. *ACM Transactions on Computer Systems*, 4(1):54–70, February 1986.

[Kenley 1986] Gregory Grant Kenley. *An Action Management System for a Decentralized Operating System.* Technical Report GIT-ICS-86/01, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia, January 1986.

[Kung & Robinson 1981] H.T. Kung and J.T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[Lamport 1978] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lampson 1981] Butler Lampson. Atomic transactions. In B.W. Lampson, M. Paul, and H.J. Siegert, editors, *Distributed Systems—Architecture and Implementation*, chapter 11, pages 247–265, Springer-Verlag, 1981. Lecture Notes in Computer Science series, number 105.

[Lindsay *et al.* 1979] B.G. Lindsay, P.G. Selinger, C. Galtieri, J.N. Gray, R.A. Lorie, T.G. Price, F. Putzolu, I.L. Traiger, and B.W. Wade. *Notes on Distributed Databases.* Research Report RJ2571, IBM, San Jose, CA, July 1979.

[Lindsay *et al.* 1984] Bruce G. Lindsay, Laura M. Haas, C. Mohan, Paul F. Wilms, and Robert A. Yost. Computation and communication in R*: A distributed database manager. *ACM Transactions on Computer Systems*, 2(1), February 1984. Also available as IBM Research Report RJ3740, January 1983, and in the Proceedings of the 9th ACM Symposium on Operating Systems Principles, October 1983.

[Liskov & Scheifler 1983] Barbara Liskov and Robert Scheifler. Guardians and actions: Linguistic support for robust distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[Liskov 1984] Barbara Liskov. *Overview of the Argus Language and System.* Programming Methodology Group Memo 40, MIT Laboratory for Computer Science, Cambridge, MA, February 1984.

[Liskov 1987] Barbara Liskov. *Highly-Available Distributed Services.* Programming Methodology Group Memo 52, MIT Laboratory for Computer Science, Cambridge, MA, February 1987.

[Liskov *et al.* 1987a] Barbara Liskov, Dorothy Curtis, Paul Johnson, and Robert Scheifler. *Implementation of Argus.* Programming Methodology Group Memo 57, MIT Laboratory for Computer Science, Cambridge, MA, August 1987. Also in the Proceedings of the 11th ACM Symposium on Operating Systems Principles, November 1987.

[Liskov *et al.* 1987b] Barbara Liskov, Mark Day, Maurice Herlihy, Paul Johnson, Gary Leavens, Robert Scheifler, and William Weihl. *Argus Reference Manual.* Programming Methodology Group Memo 54, MIT Laboratory for Computer Science, Cambridge, MA, March 1987.

[Liskov *et al.* 1987c] Barbara Liskov, Robert Scheifler, Edward Walker, and William Weihl. *Orphan Detection.* Programming Methodology Group Memo 53, MIT Laboratory for Computer Science, Cambridge, MA, February 1987.

[Lundelius 1984] Jennifer Lundelius. *Synchronizing Clocks in a Distributed System.* Technical Report 335, MIT Laboratory for Computer Science, Cambridge, MA, 1984.

[Lynch & Merritt 1986a] Nancy Lynch and Michael Merritt. Introduction to the theory of nested transactions. In *International Conference on Database Theory*, pages 278–305, Rome, Italy, September 1986.

[Lynch & Merritt 1986b] Nancy Lynch and Michael Merritt. *Introduction to the Theory of Nested Transactions.* Technical Report 367, MIT Laboratory for Computer Science, Cambridge, MA, July 1986. To appear in Theoretical Computer Science. A shortened version appears as [Lynch & Merritt 1986a].

[Lynch & Tuttle 1987] Nancy Lynch and Mark Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth ACM Symposium on Principles of Distruted Computing*, pages 137–151, ACM, August 1987. Expanded version available as Technical Report number 387, MIT Laboratory for Computer Science, Cambridge MA, April 1987.

[Marzullo 1983] Keith Marzullo. *Loosely-Coupled Distributed Services: A Distributed Time Service.* Ph.D. thesis, Stanford University, Stanford CA, 1983.

[McKendry & Herlihy 1987] Martin McKendry and Maurice Herlihy. *Timestamp-based Orphan Elimination.* Technical Report CMU-CS-87-108, Dept. of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1987.

[Mohan & Lindsay 1983] C. Mohan and Bruce Lindsay. Efficient commit protocols for the tree of processes model of distributed transactions. In *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing*, pages 76–88, ACM, August 1983.

[Moss 1981] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* Technical Report 260, MIT Laboratory for Computer Science, Cambridge, MA, April 1981.

[Mueller *et al.* 1983] Erik T. Mueller, Johanna D. Moore, and Gerald J. Popek. A nested transaction mechanism for LOCUS. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 71–89, ACM, October 1983.

[Reed 1978] David P. Reed. *Naming and Synchronization in a Decentralized Computer System.* Technical Report 205, MIT Laboratory for Computer Science, Cambridge, MA, September 1978.

[Skeen 1981] Dale Skeen. Nonblocking commit protocols. In *Proceedings of the International Conference on Management of Data*, pages 133–142, ACM/SIGMOD, 1981.

[Spector & Swedlow 1987] Alfred Z. Spector and Kathryn R. Swedlow. *Guide to the Camelot Distributed Transaction Facility: Release 1.* Computer Science Dept., Carnegie Mellon University, Pittsburgh, PA, 0.7(31)[aleph] edition, September 1987. Draft.

[Spector *et al.* 1987] Alfred Z. Spector, Dean Thompson, Randy F. Pausch, Jeffrey L. Eppinger, Dan Duchamp, Richard Draves, Dean S. Daniels, and Joshua J. Bloch. *Camelot: A Distributed Transaction Facility for Mach and the Internet—An Interim Report.* Technical Report CMU-CS-87-129, Dept. of Computer Science, Carnegie Mellon University, Pittsburgh, PA, June 1987.

[Walker 1984] Edward Franklin Walker. *Orphan Detection in the Argus System.* Technical Report 326, MIT Laboratory for Computer Science, Cambridge, MA, June 1984.

[Weihl 1984] William E. Weihl. *Specification and Implementation of Atomic Data Types.* Technical Report 314, MIT Laboratory for Computer Science, Cambridge, MA, March 1984.

[Weinstein *et al.* 1985] Matthew J. Weinstein, Thomas W. Page Jr., Brian K. Livezey, and Gerald J. Popek. Transactions and synchronization in a distributed operating system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 115–126, ACM, December 1985.

OFFICIAL DISTRIBUTION LIST

Director
Information Processing Techniques Office
Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, VA 22209

2 copies

Office of Naval Research
800 North Quincy Street
Arlington, VA 22217
Attn: Dr. R. Grafton, Code 433

2 copies

Director, Code 2627
Naval Research Laboratory
Washington, DC 20375

6 copies

Defense Technical Information Center
Cameron Station
Alexandria, VA 22314

12 copies

National Science Foundation
Office of Computing Activities
1800 G. Street, N.W.
Washington, DC 20550
Attn: Program Director

2 copies

Dr. E.B. Royce, Code 38
Head, Research Department
Naval Weapons Center
China Lake, CA 93555

1 copy